ISSN 0265-2919

80p 37

THE HOME COMPUTER ADVANCED COURSE

MAKING THE MOST OF YOUR MICRO

An ORBIS Publication

IR £1 Aus $1.95 NZ $2.25 SA R1.95 Sing $4.50 USA & Can $1.95

CONTENTS

APPLICATION

OBSTACLE COURSE We look at how an 'intelligent' robot may be programmed to move freely around a room, avoiding obstacles in its path — 721

HARDWARE

PERIPHERAL VISION A wide variety of add-ons is available for every home computer. We give you some valuable tips — 729

SOFTWARE

MODEL BEHAVIOUR As we promised last week, our series on spreadsheets continues with a look at Abacus for the Sinclair QL — 724

THE DEFECT EFFECT Deus Ex Machina is a novel game that combines elements of arcade-style games with an audio soundtrack featuring showbusiness stars — 740

COMPUTER SCIENCE

FIGURE IT OUT We discover the facilities LOGO offers for working with numbers — 735

JARGON

HANDSHAKING TO HEADER A weekly glossary of computing terms — 728

PROGRAMMING PROJECTS

NAME CALLING We develop a machine code routine for the Commodore 64 and BBC Micro to complete our search and replace program — 726

MACHINE CODE

DESIGN SENSE Structure is important in machine code, particularly with large programs. We consider some general guidelines — 738

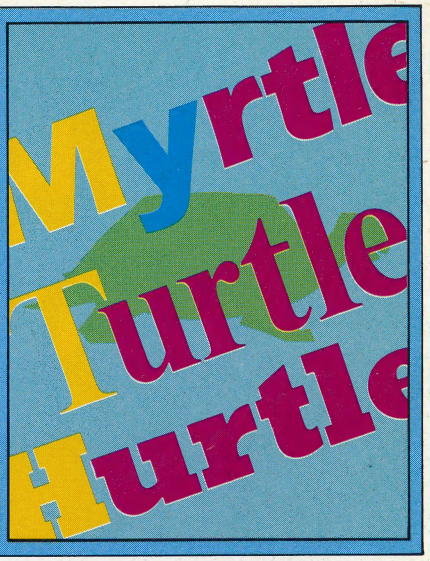
WORKSHOP

SINE WRITING Now that our D/A converter is complete, we begin to develop the software to produce sound signals — 732

REFERENCE CARD We continue to list extracts from the Z80 programmers' reference card — INSIDE BACK COVER

Next Week

- **Computer-generated poetry? In our LOGO course we develop a poetry program to demonstrate the word-handling powers of the language.**
- **We discover how robots can be programmed to perceive objects by comparing what they see with an internal model.**
- **In our 6809 machine code course we begin to develop a debugger program.**



QUIZ

1) Why was Stirling Mouse amazed?
2) What are 'tape streamers'?
3) What is a saw-tooth wave?

Answers To Last Week's Quiz

1) Demons are special event-handling routines in Atari LOGO. An event in LOGO is a user-defined condition, similar to 'interrupts' in machine code.

2) The REPLICATE command is a common feature of spreadsheet programs. It allows text and formulae to be copied from cell to cell.

3) Continuous path training is a way of programming complicated sequences of movement into robots, by having them follow a skilled human through the movements.

4) A hacker is any computer user, but usually means an amateur programmer whose chief joy is crashing other people's systems.

Editor Mike Wesley; **Art Director** David Whelan; **Technical Editor** Brian Morris; **Production Editor** Catherine Cardwell; **Art Editor** Claudia Zeff; **Chief Sub Editor** Robert Pickering; **Designer** Julian Dorr; **Art Assistant** Liz Dixon; **Staff Writer** Stephen Malone; **Sub Editor** Steve Mann; **Researcher** Melanie Davis; **Consultant Editor** Steve Colwill; **Contributors** Geoff Bains, Harvey Mellar, Mike Curtis, Steve Colwill, Chris Naylor, Tony Harrington, Jim Lennox, Steve Malone, Ted Ball; **Software Consultants** Pilot Software City; **Group Art Director** Perry Neville; **Managing Director** Stephen England; **Published by Orbis Publishing Ltd; Editorial Director** Brian Innes; **Project Development** Peter Brookesmith; **Executive Editor** Maurice Geller; **Production Controller** Peter Taylor-Medhurst; **Circulation Director** David Breed; **Marketing Director** Michael Joyce; **Designed and produced by Bunch Partworks Ltd; Editorial Office** 14 Rathbone Place, London W1P 1DE; **© APSIF Copenhagen 1984; © Orbis Publishing Ltd 1984; Typeset by Universe; Reproduction by Mullis Morgan Ltd; Printed in Great Britain by Artisan Press Ltd, Leicester**

HOW TO OBTAIN ISSUES AND BINDERS FOR THE HOME COMPUTER ADVANCED COURSE – Issues can be obtained by placing an order with your newsagent or direct from our subscription department. If you have any difficulty obtaining any back issues from your newsagent, please write to us stating the issue(s) required and enclosing a cheque for the cover price of the issue(s). **AUSTRALIA** – please write to: Gordon & Gotch (Aus) Ltd, 114 William Street, PO Box 767G, Melbourne, Victoria 3001. **MALTA, NEW ZEALAND & SOUTH AFRICA** – Back numbers are available at cover price from your newsagent. In case of difficulty, write to the address given for binders. **UK/EIRE** – Price: 80p/IR£1. Subscription: 6 months: £23.92. 1 Year: £47.84. Binder: please send £3.95 per binder, or take advantage of our special offer in early issues. **EUROPE** – Price: 80p. Subscription: 6 months air: £37.96. Surface: £31.46. 1 year air: £75.92. Surface: £62.92. Binder: £5.00. Airmail: £5.00. **MALTA** – Obtain binders from your newsagent or Miller (Malta) Ltd, MA Vassalli Street, Valetta, Malta. Price: £3.95. **MIDDLE EAST** – Price: 80p. Subscription: 6 months air: £43.94. Surface: £31.46. 1 year air: £87.88. Surface: £62.92. Binder: £5.00. Airmail: £8.31. **AMERICAS/ASIA/AFRICA** – Price: US/CAN$1.95/80p. Subscription: 6 months air: £51.74. Surface: £31.46. 1 year air: £103.48. Surface: £62.92. Binder: £5.00. Airmail: £9.44. **SOUTH AFRICA** – Price: SA R1.95. Obtain binders from any branch of Central News Agency or Intermag, PO Box 57394, Springfield 2137. **SINGAPORE** – Price: Sing $4.50. Obtain binders from MPH Distributors, 601 Sims Drive, 03-07-21, Singapore 1438. **AUSTRALASIA/FAR EAST** – Price: 80p. Subscription: 6 months air: £55.38. Surface: £31.46. 1 year air £110.76. Surface: £62.92. Binder: £5.00. Airmail: £9.84. **AUSTRALIA** – Price: Aus$1.95. Obtain binders from First Post Pty Ltd, 23 Chandos Street, St Leonards, NSW 2065. **NEW ZEALAND** – Price: NZ$2.25. Obtain binders from your newsagent or Gordon & Gotch (NZ) Ltd, PO Box 1595, Wellington.

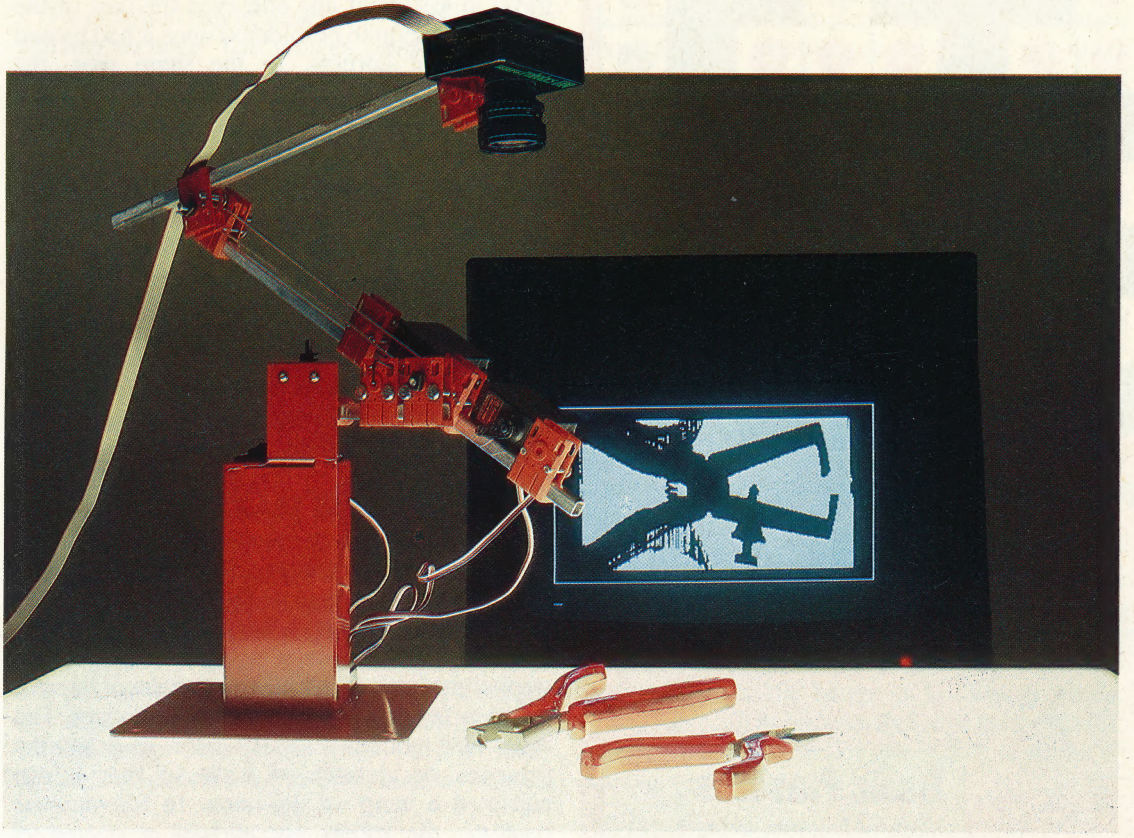
ADDRESS FOR BINDERS AND BACK ISSUES – Orbis Publishing Limited, Orbis House, Bedfordbury, London WC2 4BT. Telephone 01-379 6711. Cheques/postal orders should be made payable to Orbis Publishing Limited. Binder prices include postage and packing and prices are in sterling. Back issues are sold at the cover price, and we do not charge carriage in the UK.

NOTE – Binders and back issues are obtainable subject to availability of stocks. Whilst every attempt is made to keep the price of the issues and binders constant, the publishers reserve the right to increase the stated prices at any time when circumstances dictate. Binders depicted in this publication are those produced for the UK and Australian markets only. Binders and Issues may be subject to import duty and/or local taxes, which are not included in the above prices unless stated.

ADDRESS FOR SUBSCRIPTIONS – Orbis Publishing Limited, Hurst Farm, Baydon Road, Lambourn Woodlands, Newbury Berks, RG16 7TW. Telephone: 0488-72666. All cheques/postal orders should be made payable to Orbis Publishing Limited. Postage and packaging is included in subscription rates, and prices are given in sterling.

COVER PHOTOGRAPHY BY MARCUS WILSON-SMITH

OBSTACLE COURSE



IAN McKINNELL

Building Sight
If their processors are sufficiently sophisticated, mobile robots can learn a catalogue of archetypal objects for use with shape-recognition and pattern-matching algorithms. The Beasty robotic arm here is equipped with a Snap camera, which produces a digital picture, and the Snap software, which includes an object recognition module. Once the object has been 'seen' from different viewpoints, the arm has a reasonable chance of being able to recognise it in any position

In this series we have shown how an 'unintelligent' wheeled vehicle might be made to move under the control of either a human operator or a computer, and we have looked at the ways in which a robot arm can move 'intelligently'. Now we consider what needs to be done to design a robot that moves in a truly 'intelligent' fashion.

First of all, we do not want to control the robot by using a human operator. If the operator must watch the robot and control its every move then in many applications there would be no point at all in using a robot — the person might just as well perform the task the robot carries out. This does not, of course, apply in all situations. Robots used in bomb disposal work are human-controlled, because human expertise is still needed to guide them correctly.

There is also little point in controlling a robot via a fixed sequence of instructions stored in a computer. This would result in little more than an automaton — a device that will slavishly follow the built-in sequence regardless of circumstances. Again, there are times when such a device is useful: robot arms are often considered 'intelligent', even though they carry out a pre-programmed set of actions.

However, our definition of an 'intelligent' robot was one that would bring you an early-morning cup of tea. This cannot be human-controlled, as its function is to carry out its task before a human is awake. If this tea-bringing device is programmed with a fixed sequence of instructions, problems will arise if you move your bed or leave a pile of clothes on the floor.

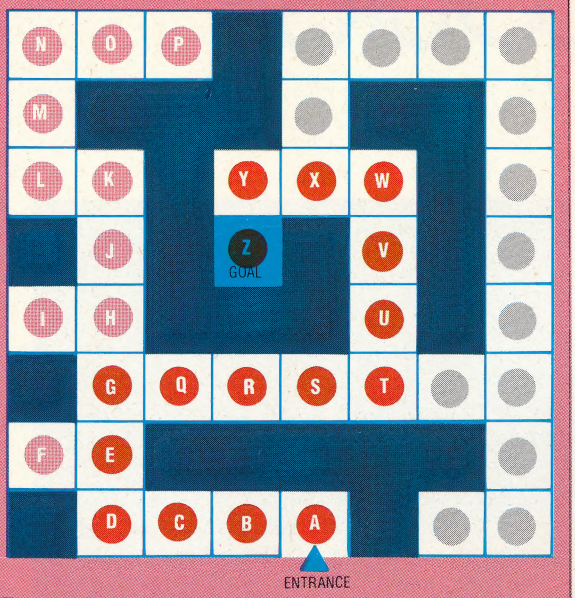
So our definition of intelligent movement is the ability of a robot to move around in its environment without being controlled by a human and without blindly following a fixed sequence of instructions. It should be able to travel from one point to another, avoiding any obstacle on the way.

There is a tradition in the field of artificial intelligence of using games of one kind or another to examine complex problems of this sort. Just as chess-playing programs have given considerable insights into other branches of artificial intelligence, so maze-running robots can help in the definition of truly intelligent movement. In the late 1970s, 'micromouse' contests began in the USA, and in 1980 the first such competition was held in Britain. The idea was very simple — a large maze some three metres square was constructed, and contestants had to design robot 'mice' that could find their way unaided to the centre. The maze consisted of small squares of equal size, the sides of which were sometimes open to show a

KEVIN JONES

Simple As ABC

Simple algorithms do work in maze-solving. This robot advances into empty spaces until it meets a dead-end — at squares F, I and P, for example. It then retreats to the last junction it encountered — square G here — marking all the intervening squares as useless in its mental map. If there are no untried routes from a junction the robot retreats to the junction before that, and so on, all the way back to the entrance, if necessary — in which case the maze is 'blind' or insoluble



Solving The Maze

```
49 REM*****COM 64**********
50 REM*  MAZE SOLVER       *
51 REM*****CBM 64**********
100 GOSUB 2000            :REM INIT
150 GOSUB 9000            :REM PR.MAZE
200 FOR L=1 TO 1
250 GOSUB 7000            :REM LOOKROUND
300 GOSUB 8000            :REM MOVE
400 NEXT L
450 RO=16:CO=12:GOSUB 9500
500 IF MZ<>HM THEN PRINT"BLIND"
550 IF MZ=HM THEN PRINT"HOME"
900 PRINT:PRINT:INPUT A$:RUN
1999 REM**********************
2000 REM*    INIT             *
2001 REM**********************
2100 SZ=10:S2=SZ*SZ:GX=SZ/2:GY=SZ+2
2120 DIM MZ(SZ,SZ),R$(4),LX(4),LY(4)
2140 X=RND(-TI)
2150 DEFFNR(N)=INT(RND(1)*N+1)
2160 HM=-1:GO=-2:WL=-42:W$=CHR$(WL)
2180 CL$=CHR$(147):H$=CHR$(19)
2200 X=GX-1:Y=GY-2:DR=3
2220 D$=CHR$(17):P$=D$
2240 FOR K=1TO5:P$=P$+P$:NEXT K:P$=H$+P$
2400 DATA 0,1,"O",-1,0,"",0,-1,"",1,0,""
2420 FOR K=1 TO 4:READ LX(K),LY(K),R$(K)
2490 NEXT K:RETURN
6999 REM**********************
7000 REM*  LOOK AROUND        *
7001 REM**********************
7110 RO=1:CO=1:GOSUB9500
7120 L=0:ND=0:FOR S=1 TO 4
7140 NX=X+LX(S):NY=Y+LY(S)
7200 IF NX<1 OR NX>SZ THEN NEXTS:RETURN
7220 IF NY<1 OR NY>SZ THEN NEXTS:RETURN
7230 MZ=MZ(NX,NY):IF MZ=0 THEN ND=S
7260 IF MZ=HM THEN ND=S:S=4:L=1
7280 NEXT S:RETURN
7999 REM**********************
8000 REM*    MOVE             *
8001 REM**********************
8100 MZ(X,Y)=DR:FL=1
8120 RO=Y+1:CO=X+1:GOSUB 9500
8140 IF ND=0 THEN ND=DR-2-4*(DR<3):FL=2
8160 PRINT R$(FL):X=X+LX(ND):Y=Y+LY(ND)
8180 DR=ND:IF Y>SZ THEN L=1:RETURN
8200 IF FL=2 THEN DR=MZ(X,Y)
8490 RETURN
8999 REM**********************
9000 REM* PRINT THE MAZE     *
9001 REM**********************
9040 PRINT CL$;:RO=1:CO=1
9050 WL=42:W$=CHR$(WL)
9060 S$="                    "
9070 FOR K=1 TO SZ+2:T$=T$+W$:NEXT K
9080 E$=W$+LEFT$(S$,SZ)+W$
9100 PRINT T$:FOR J=2 TO SZ+1
9120 PRINT E$:NEXT J:PRINT T$
9140 FOR K=1 TO S2/2
9150 WX=FNR(SZ):WY=FNR(SZ)
9160 MZ(WX,WY)=WL:RO=WY+1:CO=WX+1
9200 GOSUB 9500:PRINT W$:NEXT K
9250 CO=1+FNR(SZ):RO=1+FNR(SZ):GOSUB 9500
9300 PRINT"H":MZ(CO-1,RO-1)=HM
9350 RO=GY:CO=GX:GOSUB9500:PRINT"^"
9490 RETURN
9499 REM**********************
9500 REM* POSITION THE CRSR *
9501 REM**********************
9600 PRINT LEFT$(P$,RO)TAB(CO-1);:RETURN
```

BRIAN MORRIS

Basic Flavours

Make the following changes to this program:

BBC Micro

```
49 REM*******BBC***********
50 REM*  MAZE SOLVER       *
51 REM*******BBC***********
9040 CLS:RO=1:CO=1
9600 PRINT TAB(CO-1,RO-1);:RETURN
```

Spectrum

```
49 REM*****SPECTRUM********
50 REM*  MAZE SOLVER       *
51 REM*****SPECTRUM********
2120 DIM MZ(SZ,SZ):DIM R$(4)
2130 DIM LX(4):DIM LY(4)
2140 RANDOMIZE
2150 DEFFNR(N)=INT(RND*N+1)
9040 CLS:RO=1:CO=1
9600 PRINT AT (RO-1,CO-1);:RETURN
```

possible route, and sometimes closed to denote a wall. The mouse that reached the centre in the shortest time won the contest.

At the first British Micromouse contest, there were five entrants only. Some of these behaved in an extremely erratic fashion — one could not even travel in a straight line and even the best of the mice became quite bewildered once it had turned a couple of corners. In the same year, the European Finals of the competition were held, and mice began to arrive from Finland, Switzerland and Germany. Eventually, a mouse did succeed in negotiating the maze correctly; this was Nick Smith's 'Stirling Mouse', which was equipped with simple mechanical sensors that ran along the top of the maze walls and was powered by a simple stepper motor. Since then, interest in such competitions has grown, and in the 1984 Euromouse Contest in Madrid the fastest time to the centre of the maze was 31.4 seconds. Some contestants were still unable to reach the centre at all, but most succeeded.

MAPPING THE MAZE

So how does a robot mouse negotiate a maze? In general, the robot must have a precise method of moving itself around so that it knows its exact position at any time — this can be achieved by mounting the robot on wheels and driving it with stepper motors, often using some form of internal position feedback, such as shaft encoders. The robot also requires a set of sensors to detect the presence or absence of walls so that it can construct a 'map' of the maze. In Micromouse contests, the robots are allowed a couple of training runs, which they use to work out a plan of the course. They then make the competition run, during which they are timed in their attempts to reach the centre.

Although precise methods vary from one robot to another, one answer is to have the robot fitted with a simple tactile sensor at its front. Sitting at the centre of each square of the maze in turn, it can test to see if a wall is directly in front of it. It then turns clockwise through 90°, tests again, and repeats the sequence. Eventually it will 'know' where all the walls are in each square of the maze. This information can be stored as a single four-bit binary number — so 1111 in binary would represent a square with walls on all four sides (impossible in practice, as the robot could never enter that particular square), and 0000 would represent a square with no walls at all. 0111 would then represent a square with three walls and one opening — a cul de sac.

This information could be held in a two-dimensional array — in BASIC, DIM A(16,16) could be used to represent a maze with 16 'cells' in each direction. The robot then has to work out a route that will take it to A(8,8), if that is considered to be the centre of the maze. Often the robot has a built-in computer program that works out a tree structure for each route through the maze. Many of the branches of the tree will lead to dead ends or

bring the robot back to a point it has already visited; in these cases the branches are 'pruned' and disregarded. The program then searches along the remaining branches to find the route with the least number of squares. It then adopts that path as its route to the centre.

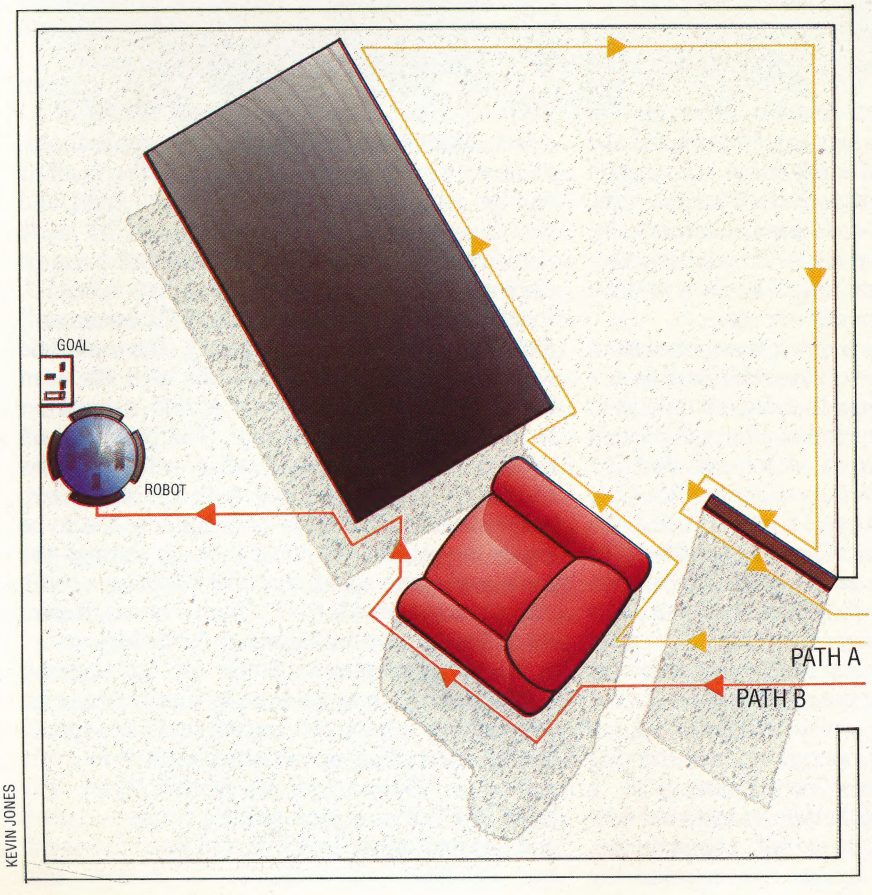
This method can be adapted to provide a more efficient strategy. The sensors on the robot are crucial to its success. For instance, simple mechanical touch sensors require the robot to actually bump into each wall to map its path; proximity sensors can detect a wall without actually touching it and a distance sensor can detect the position of a wall at the end of a long clear path in the maze. Obviously, equipping the robot with four sensors instead of just one would enable it to 'look' in all four directions at once and would remove the need to make it turn around in each square.

AROUND THE HOUSE

So we can see that a robot can act 'intelligently' as it negotiates a maze. In many respects, the problem of constructing a robot that can find its way around your home is very similar. The robot must use sensors to work out the positions of all the objects in a room, and it must then plan a route that will take it round any obstacles to its destination. The additional problems involved in this type of intelligent movement stem from the fact that a room is much more complex in design than a maze. The typical room is not neatly divided into squares, nor do all its contents remain in the same place. Your tea-making robot may learn the position of various objects — but if you move a chair, or if a cat sits on the floor, the robot must then modify its chosen path.

This problem can only be solved by having the robot make continual use of its sensors to update its internal map. The problem of the cat requires more thought because, as robots do not know anything about cats (or about people, for that matter), it is difficult for it to work out what to do at its first feline encounter. (No doubt the cat will have the same problem when it first meets a robot.) The best solution is to fit the robot with a movement sensor — which is a distance sensor that responds to variable distance measurement and can thus cope with moving objects. Once the moving object has been detected, the best thing the robot can do is to stop moving altogether until the object itself stops moving or goes away. This may not sound very intelligent, and is certainly less friendly than going up to the cat and stroking it, but such an action is very similar to the reaction shown by many animals, which 'freeze' when they detect moving objects.

The whole subject of intelligent movement is thus intrinsically linked to the use of sensors in conjunction with a computer program. A robot without sensors will not be able to move intelligently, and the more sensors a robot is equipped with, the better its knowledge of the world will be. It is this knowledge of the world that enables the the robot to exhibit signs of intelligence.



A Room Of One's Own
Finding a way around unknown objects is never easy. Path A shows the track of a simple household robot trying to find the electrical socket. Its only object avoidance algorithm (known as 'wall-following') is to follow the edges of things while its short-range sensors seek the object. This primitive method can be very successful in simple surroundings, but is susceptible to traps and pitfalls of the kind shown here.

Path B shows the track of a similar robot with a slightly improved algorithm: when it has to turn along the side of an object it prefers to turn through the smallest possible angle since this will reduce the amount of 'back-tracking' that it does. This simple change greatly reduces its vulnerability to traps and allows its scanning sensors to control its behaviour more effectively

MODEL BEHAVIOUR

PROMPTS MENU

COMMAND MENU

Our series of articles on spreadsheet modelling has so far concentrated on Psion's Vu-Calc, a simple cassette-based package for the Spectrum and BBC Micro. Here we turn our attention to Abacus — the spreadsheet package, also designed by Psion, and supplied with the Sinclair QL.

Much of the interest generated by the Sinclair QL has centred on the four software packages supplied with the machine: Quill (a word processor), Archive (a database), Easel (a graphics program) and Abacus (a spreadsheet). These packages contain some elements of integrated design (see page 502). Data may be transferred between them; spreadsheet models, for example, may be displayed as graphs or incorporated into a document prepared using Quill. The display screens are similar, and some commands are common to all the packages: for example, three of the QL's five function keys give identical results in all four applications (F1 is the Help key, F2 controls the 'prompts' area at the top of the screen and F3 calls up commands). However, the programs must be loaded and run separately.

There are two different ways of loading Abacus (or indeed any of the QL packages). The first involves putting the Abacus cartridge in Microdrive 1 and then pressing F1 to select the monitor option or F2 if a television is used as a display. The QL packages include 'boot' routines, and the program will thus load automatically. Alternatively, if the screen is already selected, enter `lrun mdv1_boot` (assuming Abacus is in drive 1), and the initial screen will appear.

The screen shows the top left-hand portion of the spreadsheet matrix — Psion refers to this as a 'grid' in its documentation. Initially, columns A to F and rows 1 to 15 are displayed, although Abacus has a maximum grid size of 64 columns and 255 rows. (Compare this to the maximum Vu-Calc grid size of 28 columns and 55 rows.) Above the grid display is a collection of prompts, and below it is a data entry line, together with information showing the status of the current model. The prompts can be removed (by pressing F2), but are very useful for beginners as they explain the choices available at any time. This is, in effect, a 'menu' area, indicating which function keys are used to control various operations, and showing how to move the cursor or go directly to a particular cell, how to enter data or text, and how to call up commands. It is not a true menu, however — options cannot be selected by positioning the cursor over the relevant choice, but must be typed in by the user.

Using Abacus for basic tasks is very simple, although for more advanced modelling some commands and expressions will take some time to get used to. The following example, again based on a home budget, will illustrate Abacus at work.

First of all, we need a general heading. As with Vu-Calc, text entries must be preceded by a double quotation mark. We will call our model CASH FORECAST and, by pressing the appropriate cursor key, we move to cell D1 and simply type a double quote followed by the text. Abacus, like most spreadsheets, allows text to 'overflow' a cell if the adjacent cell is empty, so it is easy to enter even long titles anywhere on the grid.

CASH FORECAST

We can also underline the heading, thus improving the appearance of our model. To do this, we must move the cursor to the cell below our title (D2) and enter `rept ("=",len(d1))`. Here, `rept` is the equivalent of Vu-Calc's REPLICATE command, and = tells the program which symbol to use (we are using the 'equals' sign as a double underline). The rest of the command — `len(d1)` — is a neat way of telling Abacus to repeat the symbol for the length of the text in cell D1.

Unlike Vu-Calc, which has a fixed column width of nine characters, Abacus allows us to select different widths for different applications. Here we need to make column A wider, to allow enough room to enter text of varying length, and we also require the other columns to be made narrower, so that the date for six months can be displayed. To do this, we use the function key F3, following this with G (to select the GRID command) and W (for the WIDTH command). The input line will indicate that the current width is 10. We want to change the width of column A to 15, so we enter 15 in response to the prompt. The program will now prompt for a range of cells to which the new width will apply. Entering A and A as the two parameters indicates that only column A is to be set to this width. We must then go through the same procedure again, this time selecting a width of 6 and a range of B to G. There is now sufficient room for six months' figures to be displayed.

We must now enter labels for the 'months' columns. These may be done by simply typing the relevant text in each cell, but Abacus has a special facility for calling up month names. Move the cursor to A3 and type `row=month(col()-1)`. The input line will now prompt for a range — enter B and G in reply to the prompts, and the columns will be labelled automatically. 'January' and

'February' contain too many letters to fit into our adjusted columns, so these must be abbreviated. This is done by overtyping, remembering the quote marks to indicate text.

The next step is to enter headings for the various rows, moving the cursor down a line after each entry (unfortunately, Abacus does not provide this as an automatic facility). Our model assumes that the householder is a salesman whose income is made up of both basic salary and commission. The model allows income to be calculated on the basis of commissions on forecast sales plus basic salary. Actual sales achieved can be entered as the months pass, and revised forecasts can then be made for future months. Any negative values are displayed in brackets on the finished grid. These are the headings to be entered: Sales: forecast and actual; Commission; Basic salary; Total income; Expenses: Mortgage, Rates, Water rates, Electricity, Gas, Telephone; Total expenses; Net in or (out); Opening and Closing bank balance.

After column and row headings have been entered, we can begin to fill our spreadsheet with figures. First we'll enter the sales forecasts for the six months displayed on the screen. These are £10,000, £12,000, £13,000, £11,000, £12,000 and £15,000, but should be entered without the pound sign or comma. Actual sales can't be entered yet, so we'll continue by calculating commission on the forecast sales. This is calculated as 20 per cent of sales over £10,000, so the formula to be entered into cell B10 is row=(sales-10000)*.2. Once again, the range required is B to G; enter these letters in response to the prompts and the commission will be calculated and instantly displayed.

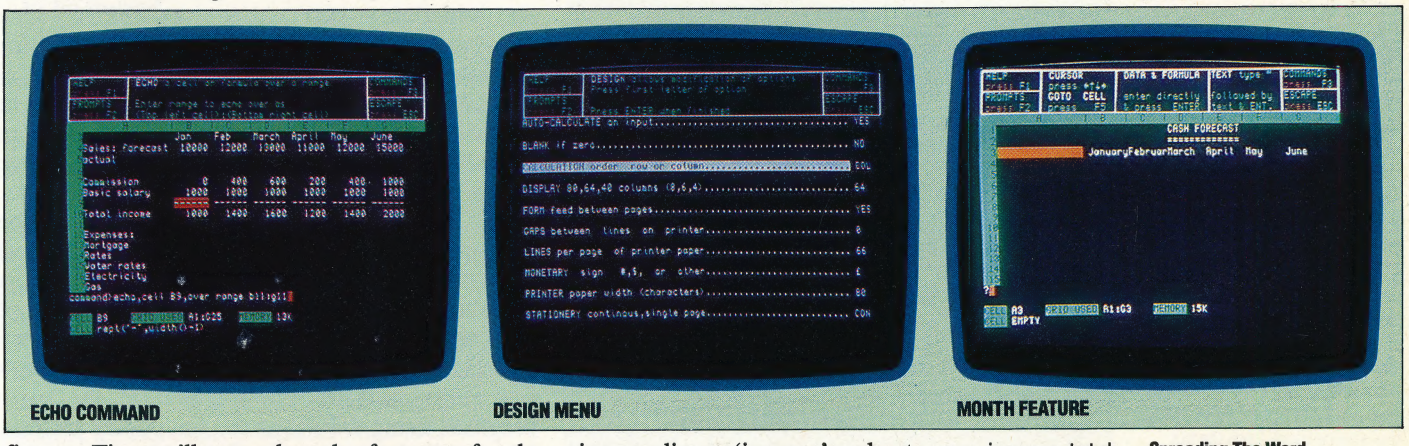
If the basic salary is £1,000 per month, this can be entered at cell B11 as row=1000, again over the range B to G. The formula for total income may now be keyed in at B13. This is row=sum(col) with a row range of 0 to 11 and column range B to G. This completes the income calculation, but the presentation of the model can be improved by adding rules above and below the 'total income' row. To do this, move the cursor to B12 and enter row=rept("-",width()-2). Abacus will respond by producing four dashes under each 'basic salary' figure. To carry out the same operation on row 14, thus producing dashes above and below the 'total income' row, the ECHO command is used. With the cursor still at B9 the system will prompt for a range over which to reproduce the underlining — respond with B14:G14. (If you forgot to leave spare rows when the headings were entered, these can be inserted by using the GRID command.) To complete the formatting, all the underlining should be 'justified right' by using the J command over the range B2 to G14 . This will ensure that all the dashes will be aligned to the figures.

Expenses may now be entered, using the row formula for standard monthly figures such as the mortgage, or by simply entering each figure into the relevant cells for occasional payments like electricity and gas. Total expenses may then be defined as the sum of all these figures, in the same way that the total income was calculated.

Similarly, the net amount in or out can be calculated as the total income minus the total expenses. Unfortunately, Abacus appears to ignore any part of a command that follows a space, so we cannot simply enter row=total income-total expenses. We must therefore rename the total income line as 'income' and enter row=income-total, again for the range B to G. Each month's net inflow and outflow will now appear on screen.

The final step is to calculate the bank balances. First enter the initial balance — an overdraft of £200, perhaps. Enter this as -200. Now calculate the closing balance, using the formula row=net+opening with the cursor at B28, over the now-familiar B to G range. This will give January's closing balance. Opening balances for other months are calculated by entering at C27 the formula row=B28. To make this work over all columns, it is first necessary to change the order of calculation from row to column by using the DESIGN command. All opening and closing balances will then be calculated, and will be recalculated immediately if any figure is changed. As it stands, our model will depict negative balances as figures preceded by a minus sign. To change this format so that brackets signify a negative figure, we use the UNITS command and reply to the prompt with B.

Our model is now complete. It should be saved by selecting the SAVE command and an appropriate file name chosen. The model may then be printed out, or different figures entered.



IAN McKINNELL

Spreading The Word
Attractive screen design, good features and a large prompt/help facility makes Abacus a pleasant and easy spreadsheet in use. Here we show the ECHO command in use copying one row to another, the DESIGN command options for formatting the spreadsheet, and the Month feature producing the month names at the press of a key. Also featured are the default screen with the Prompts menu at the top, and the blank sheet showing the Command menu

NAME CALLING

Having looked in more detail at the way a BASIC program is stored, we can now extend the variable search program to include a facility to replace one variable name by another. Here we look at the BBC Micro and the Commodore 64 versions; in the next instalment we will develop the same program for the Spectrum.

Our variable replace program is a more demanding utility than the simple search for variable names that we developed on pages 664 and 700. For this reason we need to add a machine code program. The BBC Micro's 6502 CPU and the Commodore 64's 6510 CPU have the same Assembly language, so it is a good idea to look at them together.

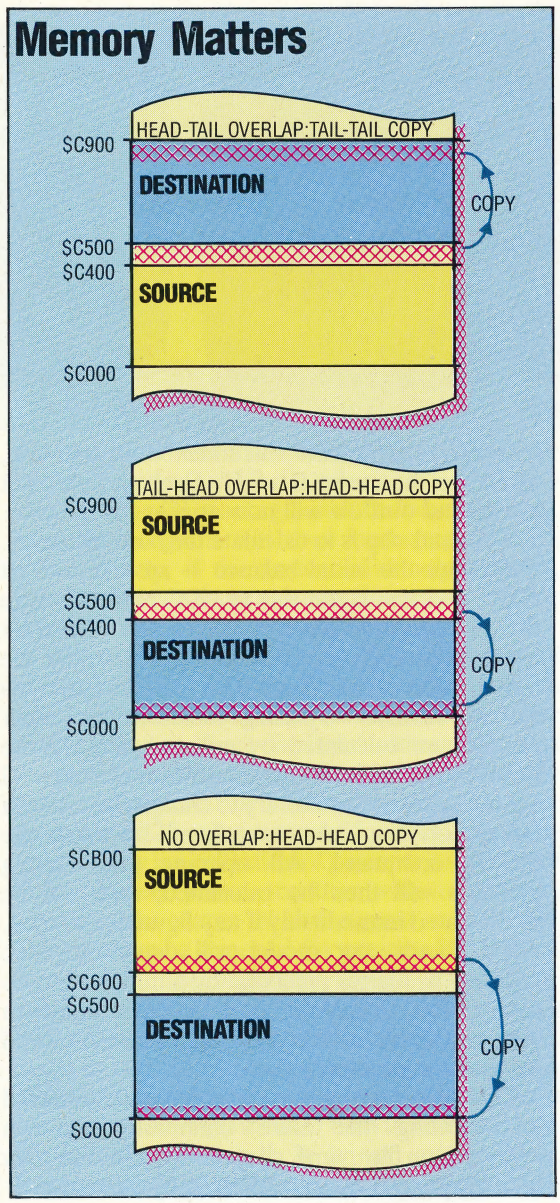
Our first task is to find a method of holding two separate programs in the computer at the same time. As we have already explained, we can do this on the BBC Micro by altering the built-in BASIC variables PAGE and HIMEM. On the Commodore 64, we need a machine code program to alter the various pointers in zero page memory. The first part of the Assembly language listing, beginning at the label SWITCH, will do this for us.

The routine SWITCH will enable us to accommodate two BASIC programs: one beginning at address 800 hex (the usual place for a BASIC program); and the other beginning at address 9000 hex. SWITCH begins by looking at the pointer TXTTAB to see which of the program areas is current, and then changes the pointer values to make the other program area current.

TXTTAB is changed to point to the start of the new program area, then FRETOP and MEMSIZ must point to the byte after the last byte of the new program area, while FRESPC points to the end of the new program area. The program then searches down the chain of link address pointers (see page 704) to find the end of the BASIC program, using VARTAB as the temporary pointer. When it finds the two zero link address bytes that mark the end of the program, it increments the previous pointer twice and copies the result into ARYTAB and STREND. In this way VARTAB, ARYTAB and STREND all point to the byte immediately after the BASIC program.

The main changes to the BASIC program that we need to make are the extra subroutines at lines 30500 and 30600. The first of these finds the end of the BASIC program, using the length of line bytes in the BBC version and the next line pointers in the Commodore 64 version.

The subroutine at line 30600 actually makes the change in the variable names. When the old and new variable names are the same length, the new name can simply be written over the old. Where the old and new variable names have different lengths, the procedure is a little more complicated. In this case, the program must either make extra space, or close up any unneeded space in the program it is changing, and make corresponding changes to the variables it uses to keep track of its position in the program being altered. It must also change the length of line byte in the BBC version and the next line pointers in the Commodore 64 version.



Intelligent Copy
The replace routine has to move large sections of the BASIC program up and down in memory when it inserts new variable names of different lengths. In this it encounters the four possible conditions of the source and destination addresses. If it always copies from the start of the source block then, when the head of the destination block overlaps the tail of the source block, the copying will overwrite some of the source data. An 'intelligent' copy routine will detect this case and avoid corruption by copying this source block from the tail first. A 'dumb' copy always copies from the head of the source block

The program also uses machine code subroutines to open or close up space. Although this could be done in BASIC, it would be unacceptably slow in even a medium-size program, with thousands of bytes to be moved every time.

There are two machine code routines: UP to make extra space; and DOWN to close up unneeded space. Details of the block of program to be moved are passed to the machine code through the memory locations CURR, LAST and DIFF.

For the subroutine UP, the following memory locations need to be initialised:

CURR: address of bottom of block to be moved
LAST: one less than address of top of block to be moved
DIFF: number of bytes to be freed

and for DOWN they need to be initialised as:

CURR: address of top of block to be moved
LAST: one less than address of bottom of block to be moved
DIFF: number of bytes to be reclaimed

Note that the two subroutines start at opposite ends of the block to be moved and make the moves in opposite directions. This is to avoid the data being overwritten before it has been moved.

To use the BBC version of the variable replace program, you first need to type in (or LOAD) and RUN the Assembly language program to put the machine code into memory. Then LOAD the program to be altered and type in:

P% = PAGE

This passes the start address of the program to the variable replacement program. Then set PAGE to a value above LOMEM, and LOAD and RUN the variable replacement program. You can get back to the altered program with:

HIMEM = PAGE — 1
PAGE = P%

To use the Commodore 64 version of the program you also need to get the machine code into memory; either with the BASIC loader program or by assembling the source code. If you assemble the source code, or load the machine code directly as machine code, you will need to POKE zeros into addresses 36864, 36865 and 36866. This is equivalent to NEWing the alternate program area.

After loading the machine code, SYS 49152 will change from one program area to the other. If you forget where you are, you can find out with PRINT PEEK (44), which will give 8 for the normal program area and 144 for the alternate program area.

Once you have the machine code in the computer, you can LOAD the program to be altered into the normal program area and the variable replacement program into the alternate program area, then RUN the variable replacement program.

C64 Switch And Copy

```
10 ADDR=49152
20 FOR LINE=1000 TO 1180 STEP 10
30 S=0
40 FOR ADDR = ADDR TO ADDR+7
50 READ BYTE
60 POKE ADDR,BYTE
70 S=S+BYTE
80 NEXT ADDR
90 READ CHECKSUM
100 IF S<>CHECKSUM THEN PRINT"DATA ERROR IN
    LINE";LINE:END
110 NEXT LINE
120 POKE36864,0:POKE36865,0:POKE36866,0
1000 DATA162,0,164,44,192,8,208,9,787
1010 DATA160,160,32,72,192,169,144,208,1137
1020 DATA7,160,144,32,72,192,169,8,784
1030 DATA162,1,134,43,133,44,134,45,696
1040 DATA133,46,160,0,177,45,170,200,931
1050 DATA177,45,224,0,208,240,201,0,1095
1060 DATA208,236,230,45,208,2,230,46,1205
1070 DATA136,16,247,165,45,133,47,133,922
1080 DATA49,165,46,133,48,133,50,96,720
1090 DATA134,51,132,52,134,55,132,56,746
1100 DATA202,136,134,53,132,54,96,160,967
1110 DATA0,177,251,164,255,145,251,160,1403
1120 DATA0,196,251,208,2,198,252,198,1305
1130 DATA251,165,253,197,251,208,234,165,1724
1140 DATA254,197,252,208,228,96,164,255,1654
1150 DATA177,251,160,0,145,251,230,251,1465
1160 DATA208,2,230,252,165,253,197,251,1558
1170 DATA208,236,165,254,197,252,208,230,1750
1180 DATA96,0,0,0,0,0,0,0,96
```

BBC Copy

```
 10 MODE 7
 20 HIMEM=HIMEM-&100
 30 CURR = &70
 40 LAST = &74
 50 DIFF = &78
 60 FOR PASS = 1  TO 2
 70 P%=HIMEM
 80[ OPT 1
 90 .UP      LDY #0
100 .UP1     LDA (CURR),Y
110          LDY DIFF
120          STA (CURR),Y
130          LDY #0
140          CPY CURR
150          BNE UP2
160          DEC CURR+1
170 .UP2     DEC CURR
180          LDA LAST
190          CMP CURR
200          BNE UP1
210          LDA LAST+1
220          CMP CURR+1
230          BNE UP1
240          RTS
250 .DOWN    LDY DIFF
260          LDA (CURR),Y
270          LDY #0
280          STA (CURR),Y
290          INC CURR
300          BNE DOWN1
310          INC CURR+1
320 .DOWN1   LDA LAST
330          CMP CURR
340          BNE DOWN
350          LDA LAST+1
360          CMP CURR+1
370          BNE DOWN
380          RTS
390]
400 NEXT PASS
410PRINT~UP,~DOWN
```

Variable Replacement

Commodore 64

Make the following changes to the program on page 700:

```
30005 INPUT "REPLACE BY"; RS
30006 CURR = 251
30007 LAST = 253
30008 DI = 255
30035 GOSUB 30500
30036 IF ERR THEN PRINT "CAN'T FIND END OF
    PROGRAM": END
30060 IF NXTLINE = 0 THEN END
30065 CURRLINE = TEXTPTR
delete line 30085
30460 IF NAME$ = T$ THEN GOSUB 30600
30465 IF ERR THEN PRINT "NO ROOM AT LINE";
LINENO: END
```

BBC Micro

Make the following changes to the program on page 665:

```
30005 INPUT "Replace by"; REPLACEMENT$
30006 CURR = &70
30007 LAST = &74
30008 DIFF = &78
30055 GOSUB 30500
30056 IF Outofroom THEN PRINT "Can't find end
    of program": PRINT "PAGE in wrong place?":
    END
30060 Textpointer = P%
30070 IF Lineno >32767 THEN END
30105 Lengthbyte = Textpointer
30460 IF Name$ = Target$ THEN GOSUB 30600
30465 IF Outofroom THEN PRINT "No room at
    line"; Lineno: END
```

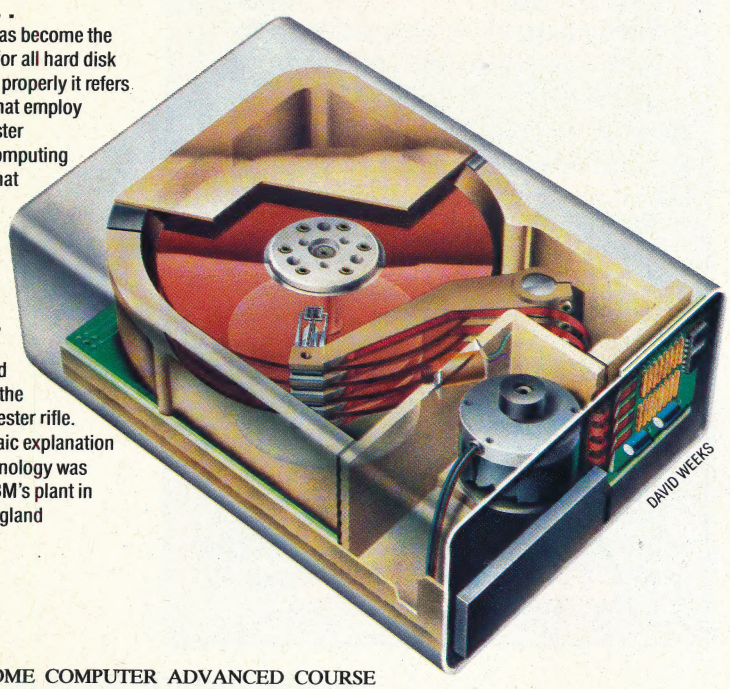
HANDSHAKING

In computing, the precise timing of operations is essential. Even such old-fashioned microprocessors as the Motorola 6502 run at a clock rate of 1 Mhz, which means that a primitive processor operation such as fetching an instruction from RAM takes about one microsecond. Remote devices such as printers and disk drives, however, operate at very much slower speeds — a dot matrix printer might take 3,000 microseconds to print one character. Transferring data between the computer and its peripherals, therefore, must be governed by strict protocol: *handshaking* is an interlocking protocol by which one device signals its readiness to receive or transmit a block of data, but does nothing until a corresponding 'ready' signal is sent by the other device. If transmissions are not controlled in some such way, data will be corrupted by the faster device reading the same data twice, or by the slower device reading only the start of incoming data.

Handshaking can be managed entirely in software, but is reasonably easy to 'hard-wire' into the device interfaces. The Motorola 6802 Peripheral Interface Adaptor (PIA), for example, communicates through its data register: when this is written to, a flag is automatically set in the PIA's status register; reading the register then resets the flag. Handshaking with this facility is reasonably straightforward: the CPU sends a character to the PIA, and continues with whatever program processes are current until it finds that the PIA's read/write flag has flipped, indicating that the external device has read the data register. This means that the CPU can send the next character to the PIA. The state of this flag, then, shows the PIA's state of readiness, and could be wired to one of the chip's pins to serve as the 'Ready/Not Ready' signal line.

HARD DISK

Hard To Say . . .
'Winchester' has become the generic name for all hard disk drives, though properly it refers only to those that employ IBM's Winchester technology. Computing legend has it that this name derives from the prototype machine's having been called a 30/30, which is the calibre and load designation of the famous Winchester rifle. The more prosaic explanation is that the technology was developed at IBM's plant in Winchester, England



An increasingly common alternative to the floppy disk drive with its interchangeable slow-speed/low capacity disk is the high-speed/high capacity *hard disk*. In this device the disk spins constantly at high speed in a sealed atmosphere — sometimes a vacuum, sometimes an inert gas such as nitrogen — and is never removed from the drive. The storage capacity of the hard disk is therefore considerably higher than that of a floppy disk, where the need for robustness, cheapness and convenience affects the engineering design and precision. Hard disk drives with between 10 and 30 Mbyte capacities, costing about the same as a modest business micro, are now freely available.

Managing the hard disk's contents requires some care — particularly the duplicating of system software and data files. Many owners 'back-up' the contents of the hard disk onto floppy disks every day, so that if a crash occurs, the floppies can be used to reconstitute the system. Copying several Megabytes can take hours and many disks, however, so high-speed tape recorders (called 'tape streamers') are often used instead of floppy disks. Even so, new software must be loaded onto the hard disk somehow — from a floppy disk drive or via a data link to another system. Either method increases the not inconsiderable cost of the device.

HASHING

When data domains are large but storage space is limited, some method of mapping the domain into the file record structure is necessary; *hashing* (see page 273) is a common method that combines efficient use of file space with reasonably high record access speeds. Let's assume that records in a file are to be arranged alphabetically according to the word in the first field of the record, and suppose that the field is 10 characters long: it would be very convenient if the ASCII characters of the word could be used to give the position in the file of the record — 'A' should go in Record 1, 'B' in Record 2, and so on. The number of possible combinations of 10 ASCII codes in the range 65 to 90 is enormous, however, and no file could accommodate them. The solution is to 'hash' the name's codes so as to produce a reasonably sized number. In this case, several different names will probably produce the same hash; when a record is to be stored but its hashed location is found to be occupied, the hash itself is rehashed to produce another possible location for the record.

HEADER

Data or programs are stored on tape or disk in files of various different formats (see page 124). The first information in the file, therefore, is written there by the system at file creation time, and consists of the file's title, type and length. This is the file *header*. If you listen to a data cassette on an audio cassette player you will hear a high-pitched tone first (the synchronisation reference tone), followed by a short burst of data followed again by the reference tone; this first section of data is the file header.

PERIPHERAL VISION

Buying a home computer is often only the first step in setting up a complete system. The basic machine may be augmented or adapted by purchasing one or more of the many peripherals on the market. Here we examine some popular add-ons, and give some tips on what to look for when buying.

Until recently, the most important peripheral for the home user was a plug-in module containing extra memory. Memory was expensive, and machines like the ZX-81 and Vic-20 were designed to keep costs to a minimum — indeed, the unexpanded ZX-81 offered the user a mere 700-odd bytes with which to write programs. Today's machines come ready-equipped with up to 64 Kbytes of RAM (some, like the QL, offer even more) and thus add-on 'RAM packs' are now rarely required. Today's user has a plethora of peripherals to choose from — modems allow communication between owners who may live hundreds of miles apart, motorised vehicles or robot arms can be controlled by using a suitable interface, and speech synthesisers may be used for fun or for educational purposes.

Although there are many different peripheral devices on the market, most of these are manufactured for the more popular machines only. This is a consideration that must be borne in mind when you decide on which machine to buy — Spectrum, Commodore and Acorn owners will always have more choice than those who opt for Orics or Sords. Newer machines take time to build up a peripheral market, although the recently introduced MSX standard will make things easier by allowing the same add-ons to be used with all machines that conform to the MSX specification.

A major consideration for the peripheral buyer is the question of compatibility — any purchase must work with any other devices that may be bought in the future. The classic example of this concerns the Sinclair Spectrum. Many Spectrum owners will have bought the Interface One and a Microdrive or two, only to find that some of their existing peripherals — and even some of their software — will not work with the Interface One in place.

However, if compatibility between devices is maintained, choosing add-ons for your machine can greatly increase the fun of computing. Suitable peripherals can allow you to design a computer system that suits your own particular requirements, and this system can then be added to as your needs change.

Storage Systems

The most common storage device for use with a microcomputer is the ordinary tape recorder. This has the benefit of being easy to use and relatively cheap, but its drawbacks soon become apparent. Programs take a long time to load, and it is difficult to keep an accurate record of what is on any particular tape. Disk drives are faster and more reliable, but cost more. Most home computers are restricted to one type of disk drive, and some of these — notably Commodore models — are notoriously slow in operation. Most home machines still use 5¼ in drives, but 3in or 3½in drives are now becoming more popular. The Oric/Atmos drive, for example, uses 3in disks with a capacity of 160 Kbytes per side. The BBC Micro is extremely flexible, allowing many different disk systems to be connected. The Torch Disk Pack effectively turns the BBC into a new computer, with a Z80 microprocessor to complement the computer's 6502 and an additional 64 Kbytes of memory. The Torch also provides four 'business' programs and comes with a version of BBC BASIC that is designed to run on the Z80 processor.

The Sinclair Spectrum, on the other hand, has no provision for the connection of standard disk drives, although some independent companies have produced special disk interfaces. Sinclair has produced the Interface One/Microdrive system, which uses a loop of tape that can hold around 85 Kbytes of data. The tape is completely under computer control, and any single item can be located within 10 seconds or so. This gives a performance that is midway between that of a tape recorder and a disk drive, at a price considerably less than the cheapest disk drive system. The Interface One has the added advantage of supplying a (non-standard) RS232 interface, and can be used for 'networking' — linking up to 64 Spectrums or QLs.

A rival to the Interface One system is the Rotronics Wafadrive. This also uses loops of tape to hold data, but includes RS232 and Centronics interfaces and a word processor program in the price. Tapes are supplied in three different sizes — 16, 64 and 128 Kbytes — with the smallest capacity tape giving the fastest working speed. A Wafadrive for the Commodore 64 is now being produced, and versions for other micros are also planned.

Shown here are the Rotronics Wafadrive, the Oric/Atmos disk drive, the Torch Disk Pack and the Sinclair Interface One with Microdrive.

Graphic Devices

Producing graphics on a home computer is made considerably easier if one of the many different types of graphics devices is used. The cheapest of these are light pens, which can be used to 'draw' directly onto the display screen by using a photoelectric cell to detect the position of the light pen's tip as it touches the screen. A development of this is the Stack Light Rifle, which can be used as an alternative to a joystick in game-playing (see also pages 230-231).

Many people find it difficult to draw 'freehand' on a display screen. Tracing lines on a flat surface is considerably easier, and many devices are manufactured to allow users to do this. Graphics tablets use a special pen that transfers any movement made on the tablet's surface to the computer; this means that they may be used to draw images freehand or to trace over printed images. Other 'drawing' devices are digital tracers, which make use of variable resistors held in a mechanical arm to detect the position of the stylus that is fixed at the tip of the arm.

Here we show British Micro's Grafpad graphics tablet, the Robot Plotter digital tracer and the Stack light pen and Light Rifle.

Modems

The development of cheap modems for home computers allows users to communicate with each other via the telephone network. A large number of modems have been produced for machines equipped with a standard RS232 interface; with the right software, these can access the Prestel database, which has many pages devoted entirely to the home user. Modems can also be used to communicate with other users via 'bulletin boards' — databases that are often run on an amateur basis by micro enthusiasts. However, the question of compatibility arises once again — different baud rates are used by different bulletin boards, and a modem that can use Prestel is often unsuitable for communication with a bulletin board.

Neither the Spectrum nor the Commodore 64 has a built-in RS232 interface, and thus cannot use standard modems. For the Spectrum, the best-selling modem is the Prism VTX5000, which has built-in software to enable the user to access Prestel. Software on tape allows two Spectrums equipped with Prism modems to exchange data programs. Commodore supplies its own modem for use with the 64, and has set up its own system, Compunet, to link Commodore 64 owners in a network.

Modem users should keep a careful eye on the clock, as enthusiasts can soon run up huge telephone bills. Fixed annual charges for Prestel and Compunet users are also high.

Our picture shows the Prism VTX5000 and the Commodore modem.

Speech Synthesisers

Many popular home machines can produce speech with the addition of a speech synthesis unit. The units available may be grouped under two headings — one type is supplied with a fixed vocabulary of 100 or so different words (the Acorn speech synthesiser for the BBC Micro uses the voice of Richard Baker), while the other uses 'allophones' — a set of different sounds and pauses from which words are constructed.

The Currah range of speech units uses the allophone system, and the company produces modules for both the Spectrum (Microspeech) and the Commodore 64 (Speech 64). Some Spectrum and Commodore 64 games, notably those produced by Ultimate, have speech built in — this is produced automatically if a Currah unit is connected. Our picture shows the Currah Speech 64, and the Cheetah Sweetalker for the Spectrum.

CHRIS STEVENS

Computer-Controlled Devices

Computers can easily be used to control devices in the 'real' world. The application usually quoted is the control of a home central heating system — this is quite possible, although hardly worth the effort as a perfectly effective time-switch is fitted to such systems anyway. Much more fun are the various wheeled vehicles that may be controlled by the computer. The Valiant Turtle is a wheeled vehicle that looks slightly like a turtle and which can produce the graphics used by the LOGO language. This can be fitted to the Spectrum, Commodore 64 and BBC Micro, and uses an infrared beam to communicate with the computer. The BBC Buggy is a similar type of device, but is linked to the computer by trailing wires. It can also be used to draw lines, and is fitted with sensors.

Our picture shows both the Valiant Turtle and the BBC Buggy.

Printer/Plotters

For anyone who uses a home computer for program development or for word processing, a printer soon becomes a vital acquisition. Most printers are either dot matrix or daisywheel in format. Dot matrix types use a grid of small dots to build up each letter, allowing graphics to be printed, and are fast in operation but produce a poorer print quality than the daisywheel, which is basically a computer-controlled typewriter.

An alternative system is the small printer/plotter that is marketed for the Tandy, Atari, Commodore and Oric computers. This uses paper that is just over four inches in width, and is fitted with four small ballpoint pens to allow multi-coloured text or graphics to be produced. The text is 'drawn' in the same way as the graphics, and a full set of characters is programmed into the device. A further alternative is provided by the Epson P40 thermal printer, which uses a column of heating elements to burn an impression on special paper. This is extremely cheap, yet gives a reasonable quality of print and runs on rechargeable batteries. Again, the paper used is fairly narrow, but the P40 can produce an 80-column printout if the condensed mode is used.

Shown here are the printer/plotter (in its Tandy/Radio Shack guise) and the Epson P40 thermal printer.

Joysticks

The first peripheral that a home computer owner buys is usually a joystick. Many computers are fitted with suitable interfaces, and some of the newer machines have joysticks supplied as standard. The most common joysticks use the nine-pin 'D'-connectors first adopted by Commodore and Atari micros, and since followed by many independent companies. BBC joysticks are decidedly non-standard, so the choice here is more limited, and Commodore has inexplicably ignored its own standard on its Plus/4 and 16 micros, restricting buyers of these machines to the new Commodore-designed joysticks.

Sinclair has recently marketed the Interface Two, a joystick interface and ROM cartridge port for the Spectrum. Until this was produced, no 'official' joystick interface had been provided for this machine, and the *de facto* standard has been the Kempston interface, the specifications of which have been adopted by many other companies. Unfortunately, the two are incompatible and many best-selling games will work quite happily with the Kempston interface but will not work at all with the Interface Two, so Kempston has now produced an interface that is compatible with software written for both Interface Two and the old Kempston format. One possible alternative is to buy a 'programmable' interface, which allows the user to run any software, whether or not it was originally designed for joystick use. This works by using the joystick to mimic the action of the keyboard and is probably the best buy for any Spectrum owner who has a large software collection.

Of the many joysticks on the market, the most unusual is the new Cheetah RAT (see also pages 590-591). This has no cable to link it to the computer, but uses an infrared beam to send and receive signals. As yet, this is available only for the Spectrum. The Amstrad system is also somewhat unusual. The Amstrad micro has a single joystick socket, but a second joystick may be connected to the first, enabling two-player games to be run.

Our joystick photograph shows (from left to right): the Amstrad joystick, the Cheetah RAT, the Kempston PRO 5000, and (front) the Kempston interface for the ZX Spectrum.

Monitors

Most home computers are used — initially at least — with an ordinary television set as the display screen. This often poses problems, as other members of the household may want to watch television while the computer owner is playing Pacman, and, anyway, the picture quality is often poor. The answer is to use a monitor, which provides a better quality picture. The user must ensure that the correct monitor is bought, as there are two main standards — RGB and composite video (see page 29). Composite video monitors are used with Atari and Commodore micros, while the BBC, Oric/Atmos and Sinclair QL machines require the RGB format, which provides the best picture quality of all. Some micros rely on the television speaker to produce sound, so these will require monitors with built-in speakers. Several television manufacturers now produce sets that are fitted with monitor interfaces; these are ideal purchases if the user requires both a television set and a high-quality display. Shown here is the Microvitec Cub.

SINE WRITING

In the last section of Workshop we built a digital-to-analogue converter to expand our user port system. We can now begin to design software to produce sound signals from this device. In this instalment we look at the production of different waveforms and discover how to determine the duration of a note.

The Project

You can monitor output either via headphones or through a stereo, using the pair of output sockets nearest the potentiometer on the D/A box. You will also need to make some simple connecting leads. For headphones: use the stereo jack plug fitted, obtain a suitable in-line socket and solder the two positive tags in this socket to a lead connected to the red socket on the D/A box. For a stereo: consult your amplifier manual to locate the audio-IN and earth connections and then make the lead. In-line stereo jack sockets are obtainable from many electrical shops. Now follow these steps: connect the buffer box and the D/A converter and plug the buffer box into the user port; plug in the headphones or stereo amp leads to the D/A sockets; turn the potentiometer on the D/A converter fully to the left and then provide transformer power to the box.

IAN McKINNELL

Once these steps have been followed, we can test the system using a short BASIC program. In essence, sound is generated electrically by providing an oscillating voltage to a speaker. We can generate a crude oscillating voltage output from our D/A converter by changing the contents of the user port data register from 0 to 255 and back in rapid succession. Type in the following program and RUN it. Turn the D/A potentiometer clockwise until sound can be heard.

```
10 REM **** CBM BASIC SOUND GENERATOR ****
20 DDR=56579:DATREG=56577
30 POKEDDR,255
35 N=1
40 POKE DATREG,0:FORI=1TON:NEXT:POKE DATREG,255:GOTO40

10 REM **** BBC BASIC SOUND GENERATOR ****
20 DDR=&FE62:DATREG=&FE60
30 ?DDR=255
35 N=1
40 ?DATREG=0:FOR I=1TON:NEXT:?DATREG=255:GOTO40
```

Notice that the BASIC program has a repeating structure, all crunched down onto a single line to produce maximum speed. There is a delay loop inserted between the data register being set to 255 and it being set to 0. The value N in line 35 sets the length of this delay. Try altering the value of N and re-running the program. You will notice that the pitch of the tone heard goes down as the value of N increases.

The highest pitch obtainable from this BASIC program will occur when the delay loop is removed altogether. Even a loop executed once has an audible effect on the pitch of the note heard.

If you have experimented with different values of N in the BASIC program given, you will have noticed that changing the value of N by 1 has a significant effect on the pitch of the note. BASIC is just not fast enough to allow us to control the rate of oscillation accurately. Instead we must use machine code.

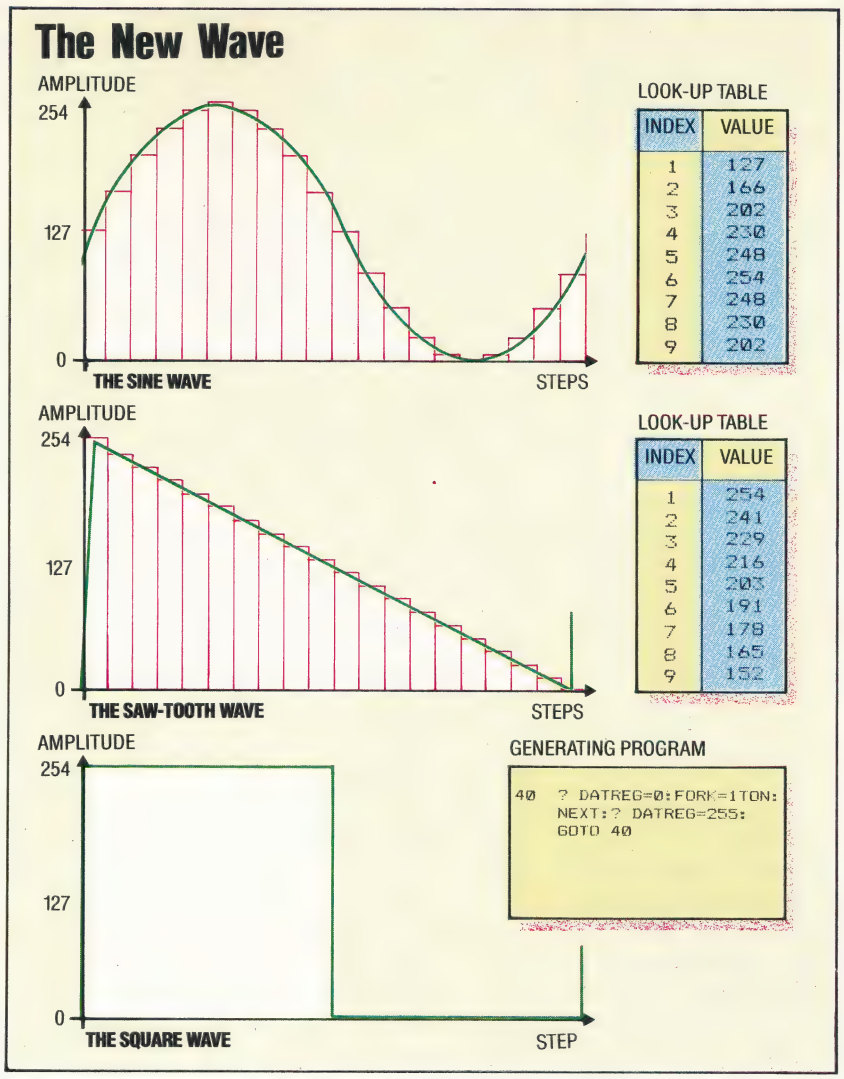
In the next instalment of Workshop we shall look at the difficult problem of controlling pitch and volume from machine code. Here we concentrate on devising a program to produce different waveforms. The waveform produced by the BASIC program used earlier was a square wave. It is, however, possible to produce other waveforms, which alter the 'quality' of the tone heard. We can digitally synthesise sine and saw-tooth waves by taking a number of samples of the waveform and putting them in a look-up table. The machine code program required to place these samples one after another in the data register is in essence very simple. Our illustration shows these three waveforms, together with accompanying look-up tables for sine and saw-tooth waves. If the waveform cycle is divided into steps, and these steps sampled, then the program loop that sends these samples out through the user port is shown left.

```
      LDX #0          2
LOOP1
      LDA TABLE,X     4
      STA PORT        4
      INX             2
      CMP #STEPS      2
      BNE LOOP1       3
     (2 if loop fails)
```

In producing sound, timing is crucial. Next to each instruction is the number of machine cycles required to execute that instruction. From this formula we can calculate the total number of machine cycles it takes to produce one complete waveform cycle: number of machine cycles = $2+(4+4+2+2+3)\times$ steps $-1 = 1 + 15 \times$ steps.

If the wave is split into 80 samples then the number of machine cycles required to produce one waveform cycle is 1,201.

As each machine cycle in 6502 code takes about a millionth of a second, the total number of waveform cycles that can be produced in one second (i.e. the frequency of the note) is given by this calculation: frequency = 1,000,000/1201 = 832 Hz. As middle C is 512 Hz, the note produced



LIZ DIXON

Form Filling

The sine wave and the saw-tooth waveforms are created by first deciding how many sample steps are to constitute one cycle of the wave. These samples of the wave's amplitude are then calculated and stored in a look-up table. The values in this table can then be copied in sequence to the user-port data register and thus to the digital-to-analogue converter where they become voltage levels. The advantage of the look-up table is that it allows the time-consuming calculations to be done in advance; actually generating the waveform therefore takes little time, and this makes a frequency range of several octaves possible. Without the look-up table the range would be restricted to two octaves.

The square wave can be generated by a BASIC program because the process is so simple. BASIC's slowness, however, restricts the frequency range considerably

should be pitched just a few notes higher than middle C.

It can be seen that the number of sample steps in which we choose to divide our waveform has a direct effect on the frequency of the final note. Doubling the number of sample steps would halve the final note frequency. Obviously, the more samples we make of the note the nearer we are likely to get to the tone quality we are synthesising, but this must always be weighed against the final maximum frequency that can be achieved for a given number of steps.

One waveform cycle is unlikely to be long enough to be audible, so we must also include code to repeat the waveform-generating section of code a set number of times. The number of repeats can be determined by setting a counter value and decrementing it to zero. To give a large range of counter values, a 16-bit number stored in two adjacent locations has been used. In addition to this code, interrupts are disabled at the start of the program by SEI and re-enabled by CLI at the end. If interrupts were to occur during execution, then this would make the timing of the program inaccurate. However, we can disable some interrupts only; non-maskable interrupts, if they occur during program execution, can still be the cause of some timing errors.

The waveform data must be set up in memory as a look-up table, with each waveform type taking 80 consecutive locations. In the Commodore version the sine wave look-up table is located in memory starting at $C000; the saw-tooth table is at $C050 and the square wave table starts at $C0A0. The machine code program is designed to default to load data from the sine wave table using indexed addressing, but we can switch to another table by modifying the program directly with a POKE from BASIC. The LDA part of LDA SINE,X is in location $C103. The start address of the table of data to be loaded has its LO–byte in location $C104 and its HI-byte in location $C105. To modify the start address of the data to be loaded all we have to do is to change the number held in $C104. Normally, for a sine wave, this location will hold 0; if we wish to change to a saw-tooth wave then all we need to do is change the contents of $C104 to 80. Changing the contents of this location to 160 will change the waveform to a square wave.

The BBC version is also designed so that the look-up tables start at the beginning of a new page in memory. Because the tables start at a new page, the HI-byte of each of the table start addresses is the same, and we need modify the LO-byte only. On a normally configured BBC Model B in mode 7, HIMEM is &7C00. By lowering the top of memory by three pages we allocate more than enough space for the look-up tables and machine code program.

FOR THE COMMODORE

```
                ;++++++++++++++++++++
                ;++++++++++++++++++++
                ;++                ++
                ;++ CBM 64  SOUND ++
                ;++   GENERATOR   ++
                ;++                ++
                ;++++++++++++++++++++
                ;++++++++++++++++++++
                ;
                PORT   = 56577       ;DATA REGISTER ADDRESS
                STEPS  = 80          ;NO. OF STEPS IN WAVE
                *=$C000
                ;
                ;++++ SET UP DATA TABLE AREA ++++
                SINE   *=*+STEPS
                SAW    *=*+STEPS
                SQUARE *=*+STEPS
                NUMBER *=*+2
                COUNT  *=*+2
                ;
                ;++++ MAIN PROGRAM ++++
                ;
78              SEI
AD F0 C0        LDA NUMBER
8D F2 C0        STA COUNT          ;SET COUNT VALUE
AD F1 C0        LDA NUMBER+1
8D F3 C0        STA COUNT+1
                ;
                LOOP2
A2 00           LDX #$00
                LOOP1
BD 00 C0        LDA SINE,X         ;GET DATA
8D 01 DD        STA PORT           ;PUSH TO USER PORT
E8              INX
E0 50           CPX #STEPS
D0 F5           BNE LOOP1          ;END OF ONE CYCLE
                ;
                ;++++ DECREMENT COUNT ++++
AD F2 C0        LDA COUNT
38              SEC
E9 01           SBC #$01
8D F2 C0        STA COUNT
AD F3 C0        LDA COUNT+1
E9 00           SBC #$00
8D F3 C0        STA COUNT+1
D0 E0           BNE LOOP2          ;IF HIBYTE>0
A9 00           LDA #$00
CD F2 C0        CMP COUNT
D0 D9           BNE LOOP2          ;IF LOBYTE>0
58              CLI
60              RTS
```

The machine code can be entered into your machine by typing in the source listing provided and assembling it to create a hex file that can be loaded whenever required. The look-up tables can be generated by running the following program:

```
900 REM **** CBM SOUND CALLING PROGRAM ****
910 :
915 DN=8:REM FOR CASSETTE DN=1
920 IFA=0 THENA=1:LOAD"5SOUND.HEX",D!,1
999 :
1000 REM **** SET UP DATA VALUES ****
1005 S=80          :REM NUMBER OF STEPS
1007 TB=12*4096 :REM START OF DATA AREA
1008 :
1010 REM **SINE WAVE **
1020 FOR I=0 TO S-1
1030 Y=127*SIN(X)+127
1040 POKE TB+I,Y
1045 X=X+2/S
1050 NEXT I
1060 :
1065 REM **** SAW WAVE ****
1070 Y=255:TB=TB+S
1080 FOR I=0 TO S-1
1090 POKE TB+I,Y
1100 Y=Y-255/S
1110 NEXT I
1120 :
1125 REM **** SQUARE WAVE ****
1130 Y=255:TB=TB+S
1140 FOR I=0 TO S/2-1
1150 POKE TB+I,Y
1160 NEXT I
1165 Y=0
1170 FOR I=S/2 TO S-1
1180 POKE TB+I,Y
1190 NEXT I
1999 :
2000 REM **** DISPLAY DATA TABLES ****
2005 TB=12*4096
2010 FORI=TB TO TB+3*S-1
2020 PRINTI,I-TB ,PEEK(I)
2030 NEXT
```

After running this program, type NEW and then type this sample program that demonstrates how to use the machine code, giving the SYS and POKE addresses required to interact with the machine code from BASIC. This program asks the user to enter the wave type required and then produces a tone each time a key is pressed.

```
10 REM ****  CBM 64 SOUND  ****
20 REM **** SAMPLE PROGRAM ****
30 :
40 DDR=56579:POKE DDR,255: REM ALL OUTPUT
65 CL=49392 :REM COUNTER LOBYTE LOCATION
67 TL=49412 :REM TYPE LOBYTE LOCATION
70 SOUND=49396:REM PROGRAM START ADDRESS
75 REM ** SET COUNTER VALUE **
80 NUM=80:NHI=INT(NUM/256):NLO=NUM-256*NHI
82 POKE CL,NLO:POKE CL+1,NHI
83 :
85 PRINTCHR$(147):REM CLEAR SCREEN
86 INPUT"WAVE TYPE (0)SINE (1)SAW (2)SQUARE";WT
87 POKE TL,WT*S
88 PRINT:PRINT"PRESS ANY KEY (RUN/STOP TO END)"
90 GETA$:IFA$=""THEN90 :REM WAIT FOR KEY
100 SYS SOUND:REM CALL MACHINE CODE
110 IF A$="X" THEN 85
120 GOTO 90
```

If you do not have an assembler or do not understand Assembly language then you can still use the machine code program by typing in this BASIC loader and running it. In this case, you can omit line 920 from the program that sets up the look-up table.

```
10 REM **** BASIC LOADER FOR CBM SOUND ****
20 REM ****        MACHINE CODE          ****
30 FOR I=49396 TO 49449
40 READ A:POKE I,A
50 CC=CC+A
60 NEXT I
70 READ CS:IF CC<>CS THEN PRINT"CHECKSUM ERROR":END
100 DATA120,173,240,192,141,242,192
110 DATA173,241,192,141,243,192,162,0
120 DATA189,0,192,141,1,221,232,224,80
130 DATA208,245,173,242,192,56,233,1
140 DATA141,242,192,173,243,192,233,0
150 DATA141,243,192,208,224,169,0,205
160 DATA242,192,208,217,88,96
170 DATA9115:REM*CHECKSUM*
```

FOR THE BBC

As the BBC has its own built-in assembler, the process of combining BASIC with machine code is substantially easier than on the Commodore 64.

```
  5 REM **** BBC SOUND PROGRAM ****
  8 MODE 7
 10 HIMEM=HIMEM-&0301
 20 MC%=HIMEM+1
 30 DDR=&FE62:? DDR=255:REM ALL OUTPUT
 40 port=&FE60:REM USER PORT DATA REG
 50 steps=80  :REM NO. OF STEPS IN A WAVE
 60 table_start=MC%
 70 PROCset_up_tables
 80 PROCmachine_code
 90 PROCsample_program
999 END
1000 DEF PROCmachine_code
1010 :      -
1020 FOR opt%=1 TO 3 STEP 3
1030 P%=MC%
1060 sine=P%:  P%=P%+steps
1070 saw=P%:   P%=P%+steps
1080 square=P%:P%=P%+steps
1090 number=P%:P%=P%+2
1100 count=P%: P%=P%+2
1110 [
1120 OPT opt%
1130 \**** MAIN PROGRAM STARTS HERE ****
1150 .sound
1160        SEI
1170        LDA number
1180        STA count
1190        LDA number+1
1200        STA count+1
1220  .loop2
1230        LDX #&00
1240 .loop1
1250        LDA sine,X
1260        STA port
1270        INX
1280        CPX #steps
1290        BNE loop1
1310 \**** DECREMENT COUNT ****
1320 \
1330        LDA count
1340        SEC
1350        SBC #&01
1360        STA count
1370        LDA count+1
1380        SBC #&00
1390        STA count+1
1400        BNE loop2
1410        LDA #&00
1420        CMP count
1430        BNE loop2
1440        CLI
1450        RTS
1455 ]
1460 NEXT opt%
1480 ENDPROC
2000 DEF PROCset_up_tables
2020 REM **** SINE WAVE ****
2025 x=0
2030 FOR I=0 TO steps-1
2040 y=127*SIN(x)+127
2050 ?(table_start+I)=y
2060 x=x+2*PI/steps
2070 NEXT I
2090 REM **** SAW WAVE ****
2100 y=255:table_start=table_start+steps
2110 FOR I=0 TO steps-1
2120 ?(table_start+I)=y
2130 y=y-255/steps
2140 NEXT I
2160 REM **** SQUARE WAVE ****
2170 y=255:table_start=table_start+steps
2180 FOR I=0 TO steps/2-1
2190 ?(table_start+I)=y
2200 NEXT I
2220 y=0
2230 FOR I=steps/2 TO steps-1
2240 ?(table_start+I)=y
2250 NEXT I
2270 REM **** DISPLAY DATA TABLES ****
2280 table_start=MC%
2290 FOR I=table_start TO table_start+3*steps-1
2300 PRINT ~I,~(I-table_start),? I
2310 NEXT I
2330 ENDPROC
3000 DEF PROCsample_program
3020 counter=MC%+3*steps: REM COUNTER LOBYTE LOCATION
3030 type=loop1+1: REM TYPE LOBYTE LOCATION
3040 count_value=80
3050 count_hi=count_value DIV 256
3060 count_lo=count_value MOD 256
3070 ?counter=count_lo
3080 counter?1=count_hi
3090 CLS
3100 INPUT"WAVE TYPE (0) SINE (1) SAW (2) SQUARE";wave
3110 ?type=wave*steps
3120 REPEAT
3125 PRINT"PRESS ANY KEY (X TO EXIT)"
3130 A$=GET$
3140 CALL sound
3150 UNTIL A$="X"
3160 GOTO 3090
```

FIGURE IT OUT

In this instalment of our LOGO course, we look at the facilities the language offers for working with numbers. LOGO would probably not be the first choice of language for applications that require a lot of calculation, but it does offer an impressive array of numerical primitives.

Almost all LOGO implementations support both integer and real (decimal) arithmetic, using the infix operators + - * / . These operators are called 'infix' because they are written between the numbers they work on — for example, 3+4. Some LOGOs also include 'prefix' arithmetic, in which our example would be written as SUM 3 4. One advantage of this notation is that it is consistent with the way in which other LOGO operations and commands are written.

MIT LOGO supports infix arithmetic only, but it is simple to program prefix forms if they are required. Define SUM and PRODUCT and try them:

```
TO SUM :A :B
  OUTPUT :A + :B
END

TO PRODUCT :A :B
  OUTPUT :A * :B
END
```

The 'precedence' of operations (the order in which they are carried out) follows the usual mathematical rules. Anything within brackets is done first, followed by multiplications and divisions, and finally additions and subtractions:

```
PRINT (3 + 4) * 5
PRINT 3 + 4 * 5
```

Now try the prefix forms:

```
PRINT PRODUCT 5 SUM 3 4
PRINT SUM 3 PRODUCT 4 5
```

This demonstrates another advantage of the prefix forms — there is no need for rules of precedence and the line is evaluated in the same way as any other line of LOGO commands.

The usual division operation (/) gives the result as a real number. Two other operations, QUOTIENT and REMAINDER, are often useful for working with integers.

```
QUOTIENT 47 5 is 9
REMAINDER 47 5 is 2
```

A standard method for converting a number in base 10 to binary is to keep dividing the number by two until the result is zero. The binary number is found by writing the remainders found at each stage in reverse order. For example, to convert 12 to binary:

```
12/2 = 6; remainder = 0
6/2  = 3; remainder = 0
3/2  = 1; remainder = 1
1/2  = 0; remainder = 1
```

So, reading the remainders upwards, we find that decimal 12 is 1100 in binary.

Using QUOTIENT and REMAINDER we can implement this technique easily in LOGO. By putting the print statement *after* the recursive call we get the remainders printed in the correct (reverse) order.

```
TO BIN :X
  IF :X = 0 THEN STOP
  BIN QUOTIENT :X 2
  PRINT1 REMAINDER :X 2
END
```

Two operations exist for rounding numbers — INTEGER and ROUND. INTEGER outputs the whole number part of a number, simply ignoring any figure after the decimal point, and ROUND rounds a number up or down to the nearest whole number.

The following procedures calculate the compound interest on an investment at a given rate of interest. In PRETTY.PRINT, INTEGER is used to get the pounds, and ROUND is used to round the pennies to the nearest whole number.

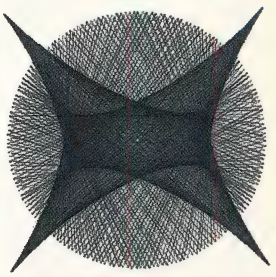
```
TO COMPOUND :PRINCIPAL :RATE :YEARS
  IF :YEARS = 0 THEN PRETTY.PRINT
  :PRINCIPAL STOP
  COMPOUND :PRINCIPAL * ( 1 + :RATE / 100 )
    :RATE :YEARS — 1
END

TO PRETTY.PRINT :MONEY
  MAKE "POUNDS INTEGER :MONEY
  MAKE "PENCE ROUND ( :MONEY —
    :POUNDS ) * 100
  ( PRINT :POUNDS "POUNDS :PENCE
    "PENCE)
END
```

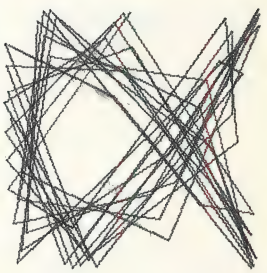
TESTING TIME

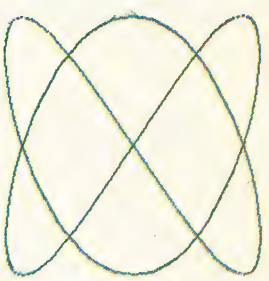
We have already used =, <, and > as logical tests in a number of procedures. The logical operations ALLOF, ANYOF and NOT can be used to combine other tests. ALLOF is true if both its inputs are true, ANYOF is true if either of its inputs is true, and NOT is true if its input is false. So we get:

```
IF ANYOF :X > 0 :X = 0 THEN PRINT "POSITIVE
IF NOT :X < 0 THEN PRINT "POSITIVE
IF ALLOF :X > 0 :X < 100 THEN PRINT
  [BETWEEN 0 AND 100]
```

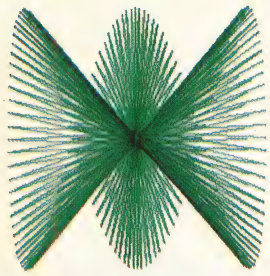
LISSAJOUS FIGURES

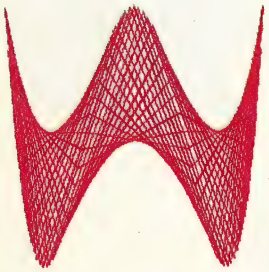






LISSAJOUS FIGURES





One Step Over The Line

The Drunkard's Walk theorem states that after N steps in completely random directions the probability is better than 0.5 that the drunkard's distance from the starting place will be less than SQR(N) steps. This is a statistical prediction based on a large number of steps, LOGO lets you test it for yourself:

```
TO DRUNKWALK :STEPNO
   :STEP
CS REPEAT :STEPNO [RT
   (RANDOM 361) FD
   :STEP]
END
```

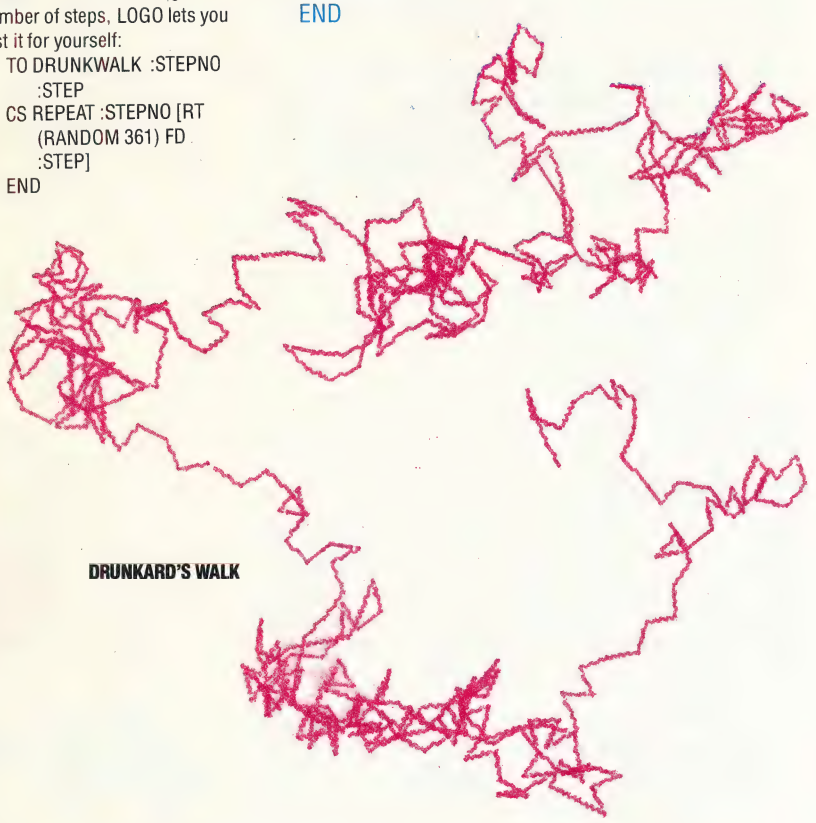
The operation NUMBER? outputs TRUE if the input is a number, otherwise FALSE is returned. We use this in the procedure PRIME?, which outputs TRUE if its input is a prime, and FALSE otherwise. It begins by checking that the input is indeed a number, and that it is greater than two. PRIME.TEST then checks to see if any integer between the square root of the number and two will divide into it exactly, leaving no remainder.

```
TO PRIME? :NO
   IF NOT NUMBER? :NO THEN PRINT [NOT A
      NUMBER DUMMY] STOP
   IF :NO < 2 THEN OUTPUT "FALSE
   OUTPUT PRIME.TEST :NO INTEGER SQRT :NO
END

TO PRIME.TEST :NO :FACT
   IF :FACT = 1 THEN OUTPUT "TRUE
   IF ( REMAINDER :NO :FACT ) = 0 THEN OUTPUT
      "FALSE
   OUTPUT PRIME.TEST :NO :FACT — 1
END
```

RANDOM NUMBERS

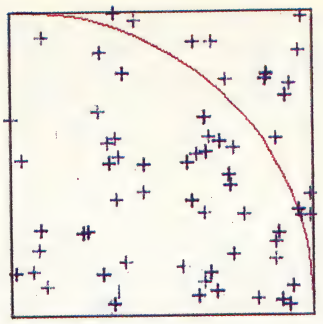
RANDOM n outputs a random integer between 0 and n–1. The procedure DRUNK makes the turtle stagger across the screen, turning a random angle at each step. The input A gives the maximum size of the turn that can be made at any time. If you run this procedure you will find that the turtle turns in vague circles, moving to the left or to the right depending on the value assigned to A.

```
TO DRUNK :A
   FORWARD 1
   RIGHT ( — :A/2 + RANDOM :A)
   DRUNK :A
END
```



DRUNKARD'S WALK

STEVE MALONE



PI COMES TO MONTE CARLO

The so-called 'Monte Carlo method' is a technique for solving mathematical problems through the use of random numbers.

We'll demonstrate by finding an approximation to pi by using this method. Our illustration shows a quarter-circle drawn within a square. The area of the square is 100 × 100 square units, and the area of the quarter-circle is (1÷4) × pi × 100 × 100 square units. The ratio of the areas circle ÷ square is equal to pi ÷ 4. Now drop a pin at random on the square 1,000 times and count how many times the pin falls within the quarter-circle; call this number IN. The value of IN/1000 should be approximately the same as the result of: circle ÷ square — i.e. pi ÷ 4. So if we do the experiment, multiply IN by four and divide by 1,000, then the result should be an approximation to pi. That is precisely what the following procedures do:

```
TO MC
   DRAW
   PU
   MAKE "IN 0
   MC1 1000 100 100
   ( PRINT [VALUE OF PI IS] 0.004 * ( :IN ) )
END

TO MC1 :NO :XNO :YNO
   IF :NO = 0 THEN STOP
   RANDOM.POINT :XNO :YNO
   IF INSIDE? THEN MAKE "IN :IN + 1
   MC1 :NO — 1 :XNO :YNO
END
```

The procedure MC simply sets the conditions, calls MC1 and prints the results. MC1 does most of the work, calls RANDOM.POINT to position the turtle, and then increments IN if the point is inside the circle. This continues until the procedure has been carried out the correct number of times.

```
TO RANDOM.POINT :XNO :YNO
   SETXY RANDOM :XNO RANDOM :YNO
END

TO INSIDE?
   IF ( XCOR * XCOR + YCOR * YCOR ) < 10000
      THEN OUTPUT "TRUE
   OUTPUT "FALSE
END
```

RANDOM.POINT sets the turtle at a random position within the square, while INSIDE? checks to see if the turtle lies within the circle. It will take some time to run this, but eventually a value for pi of 3.15999 will be obtained.

Lissajous curves are an interesting family of curves in which the x co-ordinate of each point is determined by the sine function and the y co-ordinate by the cosine:

```
TO LJ: COEFF1 :COEFF2 :STEP
   DRAW PU HT
   POS :COEFF1 :COEFF2 0 PD
   LJ1 :COEFF1 :COEFF2 0 :STEP
END

TO POS :COEFF1 :COEFF2 :ANGLE
   MAKE "X 100 * SIN ( :COEFF1 * :ANGLE )
   MAKE "Y 100 * COS ( :COEFF2 * :ANGLE )
   SETXY :X :Y
END

TO LJ1 :COEFF1 :COEFF2 :ANGLE :STEP
   POS :COEFF1 :COEFF2 :ANGLE
   LJ1 :COEFF1 :COEFF2 ( :ANGLE + :STEP ) :STEP
END
```

Logo Flavours

LCSI versions include prefix arithmetic. Atari LOGO has SUM and PRODUCT; Spectrum LOGO also has DIV and Apple LOGO has QUOTIENT, both of which correspond to MIT LOGO's QUOTIENT.
INT is used in place of INTEGER.
NUMBERP is used for NUMBER?.
The logical operators have the more usual names of AND, OR and NOT.
IF has a different syntax — IF :X = 0 PRINT "ZERO
TYPE is used in place of PRINT1.
SETPOS (followed by a list) is used for SETXY.
Use CS instead of DRAW.

Logo Exercises

1. Write a procedure to output the nth power of a number, so POWER 4 2 would output 16.
2. Write a set of procedures to convert a decimal number to hexadecimal (use a similar technique to the binary example, but this time divide by 16).
3. Write a procedure EVEN? that will output TRUE if a number is even and FALSE if it is not.
4. Use the Monte Carlo method to find the area under the curve $y=x^2$ between x=0 and x=10.

Exercise Answers

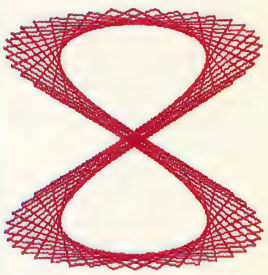
1. Convert game to using keyboard control: Change SET.DEMONS WATCH, CHECK. Delete JOYH. Add MOVE and READKEY.

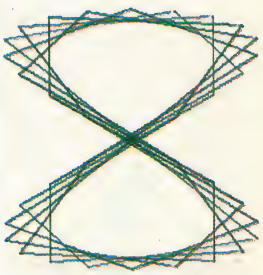
```
TO SET.DEMONS
   WHEN OVER :SHEEP1 :FENCE [SETSP 0]
   WHEN OVER :SHEEP2 :FENCE [SETSP 0]
   WHEN TOUCHING :SHEEP1:SHEEP2
      [SETSP 0]
   WHEN TOUCHING :DOG :SHEEP1 [SETSP 0]
   WHEN TOUCHING :DOG :SHEEP2 [SETSP 0]
END
TO WATCH
   MOVE READKEY
   IF :SPEED = 0 [CHECK]
   WATCH
END
TO CHECK
   IF COND OVER :SHEEP1 :FENCE [ASK
      :SHEEP1 [BK 10 RT 90]]
   IF COND OVER :SHEEP2 :FENCE [ASK
      :SHEEP2 [BK 10 RT 90]]
   IF COND TOUCHING :SHEEP1 :SHEEP2 [BUMP]
   IF COND TOUCHING :DOG :SHEEP1 [ASK
      :SHEEP1 [RT 90]]
   IF COND TOUCHING :DOG :SHEEP2 [ASK
      :SHEEP 2 [RT 90]]
   SET.SPEEDS
END
TO MOVE :COM
   IF :COM = "W [ASK :DOG [SETH 0]]
   IF :COM = "S [ASK :DOG [SETH 90]]
   IF :COM = "Z [ASK :DOG [SETH 180]]
   IF :COM = "A [ASK :DOG [SETH 270]]
   IF :COM = "Q [ASK :TURTLE [DRAW.CAGE]]
END
   TO READKEY
   IF KEYP [OUTPUT RC]
   OUTPUT "
END
```

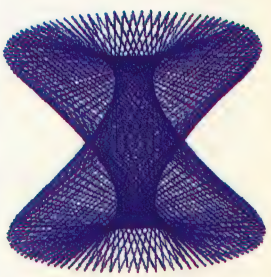
2. The meteorite game: define shape 1 as a meteorite, and shape 2 as the space ship.

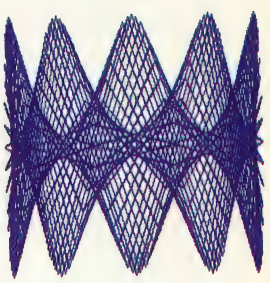
```
TO PLAY
   CS FS
   SET 0 1 [—100 80] 180 199
   SET 1 1 [0 80] 180 199
   SET 2 1 [100 90] 180 199
   SET 3 2 [0 —80] 90 50
   SET.DEMONS
   RANDOM.MOVE 0
END
TO SET :NO :SHAPE :POS :HEAD :SP
   TELL :NO SETSH :SHAPE
   PU SETPOS :POS
   SETH :HEAD ST SETSP :SP
END
TO SET.DEMONS
   WHEN TOUCHING 0 3 [BANG]
   WHEN TOUCHING 1 3 [BANG]
   WHEN TOUCHING 2 3 [BANG]
   WHEN 15 [JOYH]
END
TO BANG
   TELL [0 1 2 3]
   SETSP 0 SS
   PRINT " PRINT "
   PRINT "SPLATTERED
END
TO JOYH
   IF (JOY 1) < 0 [STOP]
   ASK 3 [SETH 45 * JOY 1]
END
TO RANDOM.MOVE :NO
   IF SPEED = 0 [( PRINT "SCORE :NO ) STOP]
   ASK RANDOM 3 [SETH 145 + RANDOM 70]
   RANDOM.MOVE :NO + 1
END
```

LISSAJOUS FIGURES

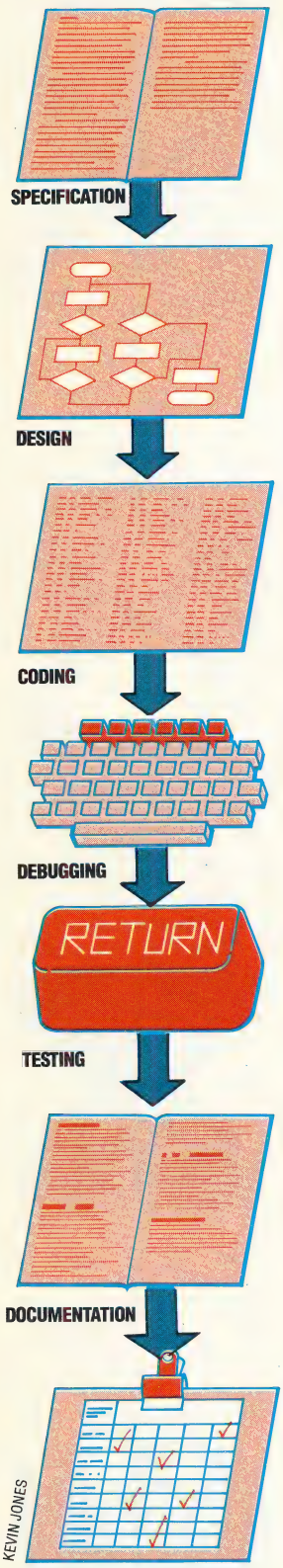








DESIGN SENSE



MAINTENANCE
Design Counts
Observing the rules of good structure is difficult in machine code programming. Developing machine code programs according to the rules of good design is not difficult, however, and pays extra dividends in clarity of design and debugging time saved

In the course so far, we have concentrated on looking at the 6809's instruction set and seeing how a few of these instructions can be put together to form simple routines. However, writing larger, more ambitious programs is a far more complex task. We consider some techniques to give structure to larger Assembly language programs.

We have talked a lot in the course about the benefits of proper program design, modular construction and structured programming in the context of high-level languages. The difficulties of programming, and the benefits of good technique, are greatly magnified at the lower level. In Assembly language, there are usually no convenient control structures, such as BASIC's WHILE. . . WEND and IF. . . THEN. . . ELSE, to enforce at least some sort of structure on the code. There are also no convenient notations, no data-typing of variables, and, to make it worse, you can expect an Assembly language program to be between six and ten times the size of a high-level program — in terms of the number of instructions. Above all, it is far easier to make errors, and these may have disastrous consequences — it is possible to wipe out all the data on a disk with an error in a single byte. To help make 6809 Assembly language programming less daunting, we consider here the most productive way to approach it.

There's nothing particularly new about structured programming or software engineering: experienced programmers have always known that forethought and clarity of approach were the ground rules for a successful programming style. What makes it seem new and original is the fact that the world of microcomputing has been largely amateur and hobbyist, but it is now becoming both more professional and more appreciative of the professional virtures. Nothing makes this point more clearly or memorably than your first attempt at debugging an undocumented, unstructured, hand-assembled machine code program that you created months ago and put aside. Good design and working methods mean good programming.

STAGES IN PROGRAM DESIGN

- Problem Specification: In this stage, the Assembly language programmer must pay particular attention to the specification of input and output. Often peripheral devices are being controlled directly — especially the keyboard and screen — so the actual signals used must be considered. There may be timing constraints as well. You may not have any convenient routines available that convert the string of bytes that come in or go out into the form in which the program reads the data — for example, converting a string of ASCII characters into a decimal number in binary form. It is important, therefore, to specify not only the form in which the data arises but also the form in which it is required by the rest of the program.
- Program Design: We must now consider the processes that will turn the program's specified input into its specified output. These should be grouped where possible into logically self-contained modules, along with the data that each process requires. There are two main techniques for 'decomposing' a program into modules: *bottom-up*, where you collect a set of what would appear to be useful modules in the context of the program and then try to fit them together; and *top-down*, where the program is successively decomposed into smaller and smaller units, concentrating on the function of each unit rather than how it is to be achieved, until the process cannot usefully be continued. Only at that point do you start considering how each module can be assembled into code.

Bottom-up design has the great advantage of using library modules, which are easy to put together, and the end result is likely to be more efficient in memory usage. The disadvantages are that the program as a whole is likely to prove more difficult to debug and test, and will not be so comprehensible. Top-down design leads to better structured programs, and each stage in the process can be tested separately by means of 'stubs', which are short routines that take the place of as yet unwritten modules by simply accepting input and providing output in the correct form without doing any processing. The disadvantages are that the programs will tend to use more memory and the routines developed are unlikely to have any immediate use elsewhere.

Within each module the data requirements, data structures and algorithms must be specified. A flowchart is useful at this level for representing algorithms, but many people find it much easier to work in a loose kind of high-level language called a pseudo-code. PASCAL is usually used as the basis for this pseudo-code, but there is no reason why BASIC cannot be used. This enables us to design algorithms and data in a way that is familiar to us, and confines the lower-level work to the relatively simple task of translating the algorithm from pseudo-code into Assembly language. This is much easier than trying to design and code in Assembly language at the same time.

- Coding: If the routines have been well designed

then this stage will probably be the easiest and least time-consuming of all. In order to translate from a high-level algorithm to low-level code it is essential that the control structures used at high level are carried over to the low level, avoiding the temptation to use BRA and JMP indiscriminately. Remember that any time you save by writing unstructured code is certain to be 'clawed back' in a frustrating trial-and-error debugging stage. In the diagram we give some examples of the way in which the common control structures can be coded — assuming, for simplicity, that the data items used are eight-bit.

One problem with coding with control structures in this way is that the program is longer than it might be. Where space is not limited then there is no point in trying to save it; short code does not usually mean shorter running times but it does mean longer development and debugging times. Where space is limited, then it is better to write in a spacious structured way, and introduce a further stage of optimisation where the working code can be shortened to take into account particular circumstances, retaining as far as possible the essential structure.

- Debugging: At this stage, each module is separately tested — using stubs where necessary — to make sure it gives appropriate outputs for valid inputs. Debugging Assembly language programs differs considerably from BASIC program debugging. To be able to see what is happening, it is necessary to be able to inspect the contents of the registers and the memory locations used by the program, and to change them if necessary. It is nearly impossible to debug an Assembly program without the use of a utility for setting and removing breakpoints. These enable you to run the program up to the next breakpoint, then dump the registers, and inspect and change memory contents.
- Testing: Once each module has been tested and debugged then the entire program has to be put together and tested with appropriate data. This is much easier when you know that all the component parts are working properly.
- Documentation: Assembly language programs are more difficult to understand than high-level programs, so documentation is even more important. In particular, it is vital to document the use of memory, the use of the stack (especially while passing parameters), and the register usage within subroutines.
- Maintenance: If a program is to be used over a period of time then at some point it will probably need revision — either to remove any bugs that appear or to make improvements. It is at this stage that time spent in careful design and documentation really pays off. If the program is badly designed and/or poorly documented then you are better off doing a complete rewrite rather than attempting to make alterations.

Now we need a project to apply these design skills to: for our first venture in structured Assembly language programming nothing could be more appropriate than a machine code monitor/debugger. If you've used an assembler before, then you may be familiar with the kind of utilities to expect from a monitor/debugger. Essentially, it gives the machine code programmer the kind of editing facilities that the BASIC programmer takes for granted — namely, the ability to inspect and change the contents of memory.

In the next instalment of the course we will take this project through the design and development stages described in this article, to create an important and extremely useful programming aid.

Basic Backbone
There are no control structures written into Assembly language, so it pays to import tried and tested methods from high-level languages. The structures shown here are clear and graceful in both high- and low-level languages, and should be used to the exclusion of all alternatives

Control Structures

The IF...THEN...ELSE Structure

Pseudo-code	Assembly Language
	THREE FCB 3
	.....
IF NUM1 = 3	IF LDA NUM1
THEN	CMPA THREE
routine1	BNE ELSE
	THEN .....
ELSE	* routine1
routine2	BRA ENDIF
ENDIF	ELSE .....
	* routine 2
	ENDIF .....

The WHILE...WEND Structure

Pseudo-code	Assembly Language
WHILE NUM1 <= 3	WHILE LDA NUM1
repeated routine	CMPA THREE
WEND	BGT WEND
	* repeated routine
	BRA WHILE
	WEND .....

The REPEAT...UNTIL Structure

Pseudo-code	Assembly Language
REPEAT	REPEAT .....
repeated routine	* repeated routine
UNTIL NUM1 < 3	LDA NUM1
	CMPA THREE
	BGE REPEAT
	UNTIL .....

The FOR...NEXT Structure

Pseudo-code	Assembly Language
FOR NUM1 = 1 TO NUM2	LDA NUM2
repeated routine	FOR .....
NEXT NUM1	*repeated routine
	DECA
	BGT FOR
	NEXT .....

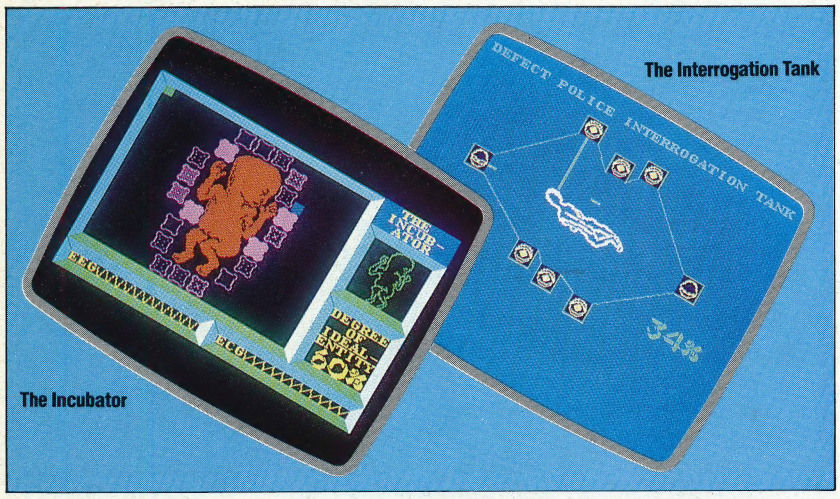
THE DEFECT EFFECT

Automata's Deus Ex Machina claims to offer 'a completely new form of computer entertainment'. Combining elements of well-known arcade games with an audio soundtrack featuring showbusiness stars, this complex program allows you to take the leading role in a 'fully animated televised fantasy'.

As computer games have developed into a major part of the leisure industry, it was perhaps inevitable that software houses would join forces with other segments of the entertainment business. Automata Software, best known for its series of games featuring the Piman, has taken the first steps in this direction by developing a product that contains not only computer software but also an audio cassette that can be synchronised with the computer program to provide a soundtrack to the game. This soundtrack features well-known figures like Jon Pertwee and Frankie Howerd.

The idea behind Deus Ex Machina, which took six months to develop and three months to program, is that an all-powerful computer of the future rebels and assists in the creation of a human 'defect'. The player, as the defect, passes through various stages in the game that depict experiences from childhood to old age. The player's involvement begins at conception by guiding the sperm towards the egg. As the child grows, it is under constant attack by the 'defect police'. It is up to the player to deflect these attacks, using either the keyboard or the joystick. Scoring is achieved by maintaining the 'ideal entity percentage', which begins at 99 per cent and drops under the assaults of the defect police. As the defect grows to adulthood, the nature of these attacks changes and the player must adapt to meet them.

The Game Progresses
Deus Ex Machina can be either played or viewed as an entertainment. There is a wide variety of screens in the game, although some do bear a resemblance to others. The tactics required to maintain the 'ideal entity' are changed constantly. The score is shown as a percentage in the bottom right-hand corner and slowly falls as the game — and the defect's life — progresses



The Interrogation Tank

The Incubator

IAN McKINNELL

Once the game is loaded, the audio soundtrack should be synchronised with the program. Care must be taken when this is done, as the various screens are timed to coincide precisely with the words and music, and this adds greatly to the enjoyment of the package.

The program is divided into two segments, one half on each side of the cassette, making up a total of 96 Kbytes of code. At the end of side one, after an amorous scene in which the player must move a cursor around the body to meet the kisses drifting towards it, the second side must be loaded. The computer should not be switched off, and again care must be taken to synchronise the soundtrack correctly. Player involvement in the second half consists mainly in jumping over obstacles before reaching 'old age'. At this point, large blood clots appear onscreen, which must be broken up by the player. At the end of the game, no matter what the score, the defect dies.

Deus Ex Machina is unusual in that there is no winning score, and in fact the player need not even participate in the game at all. Events will unfold in the same way without any participation, so you have the choice of becoming involved in the game or sitting back and watching it as a piece of entertainment. The graphics are uniformly excellent and imaginative. Although none of the screens is breathtaking, they do reflect the care and attention to detail devoted to the whole package.

The soundtrack music was written and performed entirely by Automata's co-founder Mel Croucher, who also wrote the story. The songs themselves are pleasant but not exceptional. The best number accompanies the scene in which the defect comes to life, and is sung by Ian Dury.

The story and soundtrack are quite different from most computer games and reflect the non-violent philosophy behind all Automata's computer games. Games enthusiasts who enjoy destroying fast-moving barrages of attacking aliens would probably be disappointed, and many people may find the semi-mystical content of the lyrics irritating. However, Automata should be heartily applauded for their innovative idea. The program is a bold experiment and will no doubt be considered an important step in the development of computerised entertainment.

Deus Ex Machina: For 48K Spectrum, £15.00
Publishers: Automata Ltd, 27 Highland Road, Portsmouth, Hants, PO4 9DA
Authors: Mel Croucher, Andrew Stagg
Joysticks: Optional
Format: Cassette

DATABASE

Here, courtesy of Zilog Inc., we produce another part of the Z80 programmers' reference card.

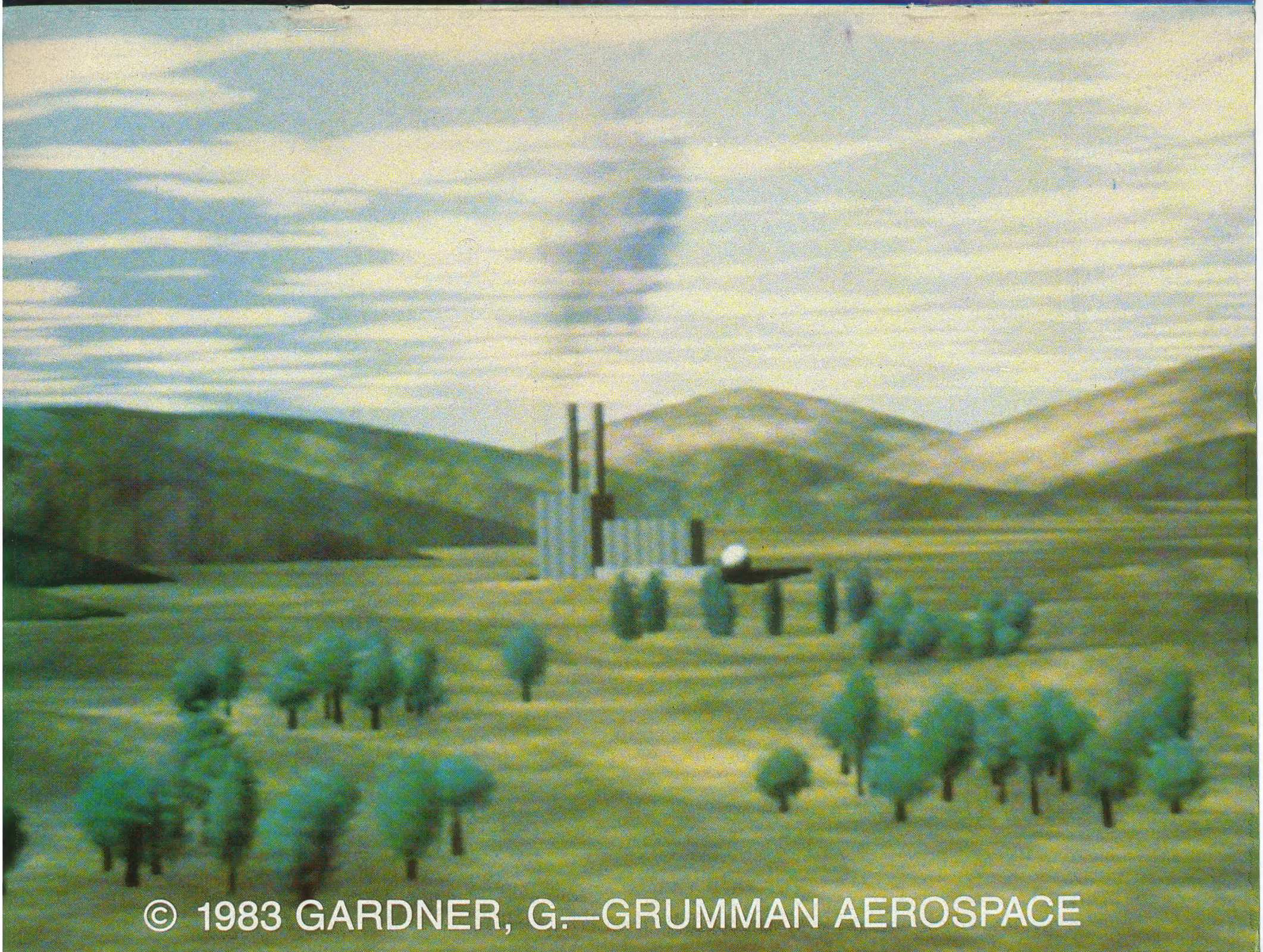
16-Bit Arithmetic Group

		SOURCE					
		BC	DE	HL	SP	IX	IY
DESTINATION 'ADD'	HL	09	19	29	39		
	IX	DD 09	DD 19		DD 39	DD 29	
	IY	FD 09	FD 19		FD 39		FD 29
ADD WITH CARRY AND SET FLAGS 'ADC'	HL	ED 4A	ED 5A	ED 6A	ED 7A		
SUB WITH CARRY AND SET FLAGS 'SBC'	HL	ED 42	ED 52	ED 62	ED 72		
INCREMENT 'INC'		03	13	23	33	DD 23	FD 23
DECREMENT 'DEC'		08	1B	2B	3B	DD 2B	FD 2B

Mnemonic	Symbolic Operation	Flags S	Z	H	P/V	N	C	Opcode 76 543 210	Hex	No.of Bytes	No.of M Cycles	No.of T States	Comments
ADD HL, ss	HL ← HL + ss	•	•	X X X	•	0	↕	00 ss1 001		1	3	11	ss Reg 00 BC
ADC HL, ss	HL ← HL + ss + CY	↕	↕	X X X	V	0	↕	11 101 101 01 ss1 010	ED	2	4	15	01 DE 10 HL 11 SP
SBC HL, ss	HL ← HL − ss − CY	↕	↕	X X X	V	1	↕	11 101 101 01 ss0 010	ED	2	4	15	
ADD IX, pp	IX ← IX + pp	•	•	X X X	•	0	↕	11 011 101 01 pp1 001	DD	2	4	15	pp Reg 00 BC 01 DE 10 IX 11 SP
ADD IY, rr	IY ← IY + rr	•	•	X X X	•	0	↕	11 111 101 00 rr1 001	FD	2	4	15	rr Reg 00 BC 01 DE 10 IY 11 SP
INC ss	ss ← ss + 1	•	•	X • X	•	•	•	00 ss0 011		1	1	6	
INC IX	IX ← IX + 1	•	•	X • X	•	•	•	11 011 101 00 100 011	DD 23	2	2	10	
INC IY	IY ← IY + 1	•	•	X • X	•	•	•	11 111 101 00 100 011	FD 23	2	2	10	
DEC ss	ss ← ss − 1	•	•	X • X	•	•	•	00 ss1 011		1	1	6	
DEC IX	IX ← IX − 1	•	•	X • X	•	•	•	11 011 101 00 101 011	DD 2B	2	2	10	
DEC IY	IY ← IY − 1	•	•	X • X	•	•	•	11 111 101 00 101 011	FD 2B	2	2	10	

NOTES: ss is any of the register pairs BC, DE, HL, SP.
pp is any of the register pairs BC, DE, IX, SP.
rr is any of the register pairs BC, DE, IY, SP.

Flag Notation: • = flag not affected. 0 = flag reset. 1 = flag set. X = flag is unknown.
↕ = flag is affected according to the result of the operation.


© 1983 GARDNER, G.—GRUMMAN AEROSPACE